

# Resumen de Sistemas Operativos

Guido Tagliavini Ponce

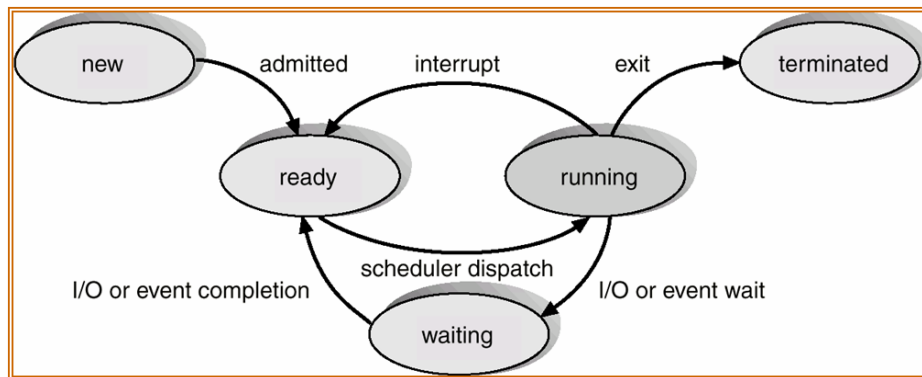
Última modificación: 12 de mayo de 2016

## Introducción

- Multiprogramación: varios programas cargados en memoria al mismo tiempo. Podemos ejecutar uno atrás del otro sin espera.
  - Aumenta el throughput.
- Multitasking (o time sharing): Varios programas se turnan para utilizar el procesador.
  - Aumenta el tiempo de respuesta.
- SO: pieza de software que hace de intermediario entre HW y los programas de usuario.
- Elementos de un SO:
  - Drivers: se encargan de la interacción entre el kernel y los dispositivos.
  - Kernel: parte central del SO.
  - Librerías de sistema: librerías y utilidades que permiten a las aplicaciones interactuar con el kernel. Por ejemplo, la librería de C.
  - Utilidades del sistema: programas que realizan tareas específicas, como por ejemplo configurar una parte del sistema o escuchar conexiones de red entrantes.
  - Intérprete de comandos: programa que permite la interacción entre el kernel y un usuario.
  - Programas de usuario.
- Proceso: un programa en ejecución. Constituido por:
  - Un identificador (pid).
  - Espacio de memoria asociado.
  - Otros atributos, como su tabla de archivos abiertos y su prioridad (ver contenido de PCB).
- Microkernel: kernel minimal, en el sentido que minimiza la cantidad de código que corre con privilegios de kernel. Se limita a MMU, IPC e IO básica.
  - El resto de los servicios se implementan afuera del kernel.
  - Es lo opuesto a un kernel monolítico.

# Procesos

- Representación en memoria:
  - Sección de texto (código de máquina del programa).
  - Sección de datos (variables estáticas y globales).
  - Heap.
  - Stack.
- De su estado también forman parte todos los componentes que se almacenan en su PCB (Process Control Block).
- La PCB contiene (entre otras cosas):
  - pid (process id).
  - Estado de scheduling del proceso (ver siguiente punto) y prioridad.
  - PC (Program Counter).
  - Estado de los registros de CPU asociados al proceso.
  - Información de la MMU: tabla de páginas, tabla de segmentos.
  - Información de IO: lista de archivos abiertos del proceso.
- Estado de un proceso:



- Control de admisión: el SO debe decidir si admitir o no un proceso. En otras palabras, si ponerlo o no en la cola de procesos *Ready*.
  - Debe regular la carga del sistema.
  - Uso de políticas de admisión, basadas en la métrica Multiprogramming Level (MPL).
- Scheduler: se encarga de decidir a qué proceso le toca usar la CPU.
- Context switch: intercambio del proceso utilizando la CPU.
  - Pasos:
    - Guardar el contexto del proceso actual en su PCB.

- Agregar un puntero a esta PCB a la cola del scheduler correspondiente.
  - Elegir un nuevo proceso, tomando una PCB de la cola ready.
  - Cargar el contexto con la información de la PCB del nuevo proceso.
  - Ejecutar la nueva tarea.
- Syscall: llamada al sistema.
  - Ver ejemplos más abajo.
  - Pasos:
    - Un proceso (kernel o usuario) realiza una llamada indicando el número de la interrupción.
    - El procesador busca la rutina asociada al número de syscall indicado.
    - Comienza la ejecución de la rutina asociada, con privilegios de kernel.
    - Se guarda el estado actual (todo lo que se vaya a pisar).
    - Al finalizar, restablece el estado y devuelve el control de la ejecución a la rutina que interrumpió (en general, con una instrucción especial, como iret). El nivel de privilegio vuelve al anterior.
- Creación de procesos:
  - La única forma es a través de la syscall fork() (ver syscall fork() más abajo).
  - El proceso init (pid 1) es el proceso padre para todos los procesos usuarios.
  - Tabla de procesos: contiene información sobre todos los procesos en ejecución.
    - La podemos ver con
 

```
ps -Al
```
    - Cada entrada está asociada a un proceso y contiene, entre otras cosas:
      - pid
      - Owner
      - Prioridad
      - ppid (parent pid)
  - Al hacer fork(), un proceso puede quedarse esperando que termine la ejecución de su hijo mediante wait(), o bien continuar su ejecución.
  - Cuando un proceso hijo termina, y el padre continúa ejecutándose pero no hizo wait(), la entrada del hijo no es eliminada de la tabla de procesos.

- Zombi: proceso que está muerto pero sigue en la tabla.
  - Cuando el padre haga wait(), el hijo será borrado de la tabla.
- Si el padre termina sin hacer wait(), la entrada del hijo se quedará esperando en la tabla.
  - Huérfano: proceso que está en la tabla pero su padre está muerto.
  - Linux asigna a init como el nuevo padre de un proceso huérfano.
  - init ejecuta wait() periódicamente para permitir la salida de huérfanos de la tabla.

## API del SO

- POSIX: Portable Operating System Interface; X: UNIX
- Especifica interfaces para varios servicios:
  - Creación y control de procesos
  - Pipes
  - Señales
  - Operaciones de archivos y directorios
  - Biblioteca C
  - Excepciones, errores de bus, instrucciones IO.

### *Creación de procesos*

- fork():
  - Crea un proceso hijo.
  - El hijo adquiere un espacio de memoria propio, con una copia del código y la memoria del padre. También recibe una copia de otras estructuras, como por ejemplo la tabla de archivos abiertos.
  - La ejecución del hijo continúa desde el mismo lugar de la llamada a fork().
- vfork():
  - Similar a fork, pero el hijo usa el espacio de memoria (código y datos) del padre. El padre queda suspendido.
  - Utilizado cuando se va a llamar a execv() inmediatamente.
- execv():
  - Copia el código de máquina de un programa en el espacio de memoria del proceso, y lo ejecuta. Destruye el espacio de memoria del programa, excepto aquellas partes compartidas.
- wait():
  - Bloquea el proceso hasta que termine de ejecutarse un (solo uno) proceso hijo.
- waitpid():
  - wait() pero especificando un proceso en particular.

- `clone()`:
  - Creación de proceso hijo, pero con capacidad para elegir cuáles estructuras de datos compartir (información del file system, espacio de memoria, signal handlers, archivos abiertos, etc.).

### *Manejo de archivos*

- `open(path, flags)`:
  - Abrir un archivo de nombre path. Devuelve un file descriptor.
- `read(fd, buffer, count)`:
  - Lee el archivo asociado al file descriptor fd.
- `write(fd, buffer, count)`:
  - Escribe el archivo asociado a fd.
- `lseek(fd, offset, whence)`:
  - Actualiza la posición de lectura sobre el archivo asociado a fd.
  - whence puede ser `SEEK_SET`, `SEEK_CUR` o `SEEK_END`.
- `close(fd)`:
  - Cierra el archivo asociado a fd.

### *Pipes*

- `pipe(int buffer[2])`:
  - Crea los dos extremos de un pipe.

### *Memoria compartida*

- `shm_open()`:
  - Crea un objeto de memoria compartida, lo cual genera una entrada en la tabla de archivos abiertos.
- `ftruncate()`:
  - Determina el espacio para ese objeto de memoria.
- `mmap()`:
  - Mapea un objeto de la tabla de archivos abiertos, en la memoria virtual de un proceso.
  - En otras palabras, un proceso que hace mmap sobre un objeto, podrá escribir en su espacio de memoria para modificar el objeto.

### *Pasaje de mensajes*

- `msgget()`:
  - Crea una casilla de mensajes.
- `msgsnd()`:
  - Envío de mensaje.
- `msgrcv()`:
  - Recepción de mensaje.

### *Sockets*

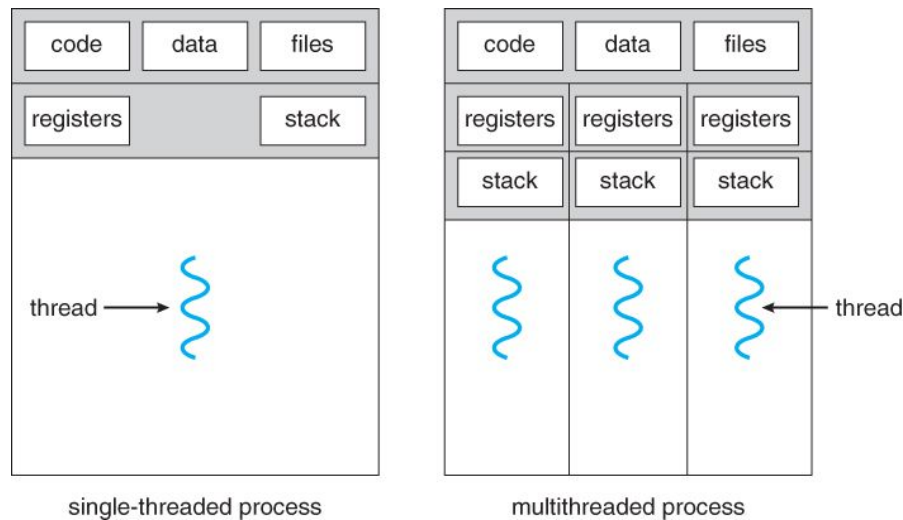
- `socket()`:
  - Crea un socket (extremo de una comunicación).
- `bind()`:
  - Asigna una dirección al socket.
- `listen()`:
  - Pone al socket a escuchar conexiones entrantes.

## Señales

- `signal(signal, handler)`:
  - Permite instalar una señal en un proceso, asociándole un handler determinado.
- `kill(pid, signal)`:
  - Envía una señal a un proceso.
- Tanto los archivos, como los pipes, como la memoria compartida y como los sockets, se leen y escriben con sus file descriptors a través de `read()` y `write()`.
- Los mensajes son un poco más sofisticados, en el sentido de que trabajan con objetos mensajes, y no como simples streams de bytes en los cuales no hay límites entre un envío y el siguiente.
- Hay IPC sincrónico (o bloqueante) y asincrónico (o no bloqueante).
  - Sincrónico: la llamada a `write()` no termina hasta que el emisor no hace `read()`. En general implica bloqueo del emisor.
  - Asincrónico: la llamada a `write()` termina, aunque el emisor no haga `read()` en ese momento. Libera al emisor para que pueda seguir haciendo otras tareas.
  - Por ejemplo, se puede usar un pipe en forma bloqueante o no bloqueante. El modo se setea a través de la función `fcntl()`.

## Threads

- Thread: unidad de uso de CPU. Constituido por:
  - Un identificador (`tid`).
  - Un PC.
  - Un set de registros.
  - Un stack.
- Proviene de un proceso, y comparte sus elementos con otros threads:
  - Código.
  - Variables estáticas y globales.
  - Heap.
  - Otros recursos, como archivos abiertos y señales instaladas.



- ¿Para qué threads si ya teníamos procesos?
  - Como comparten recursos entre ellos, son más baratos de crear que los procesos.
  - En particular, comparten el heap, con lo cual resultan una buena opción que usar `fork()` y poner a compartir memoria entre procesos.
  - Si un proceso debe hacer tanto IO intensivo como CPU intensivo, y ambas tareas son, en algún grado, independientes, podemos tener dos threads distintos, uno para IO y otro para CPU. Esto hace que cuando el thread para IO esté esperando, el thread CPU pueda aprovechar el tiempo y seguir trabajando.
  - En arquitecturas con varios procesadores (o núcleos), podemos ejecutar los threads de un mismo proceso en distintas CPUs. Esto hace que un proceso multi-threaded sea efectivamente más rápido que un single-threaded.

## Scheduling

- Aspectos de optimización:
  - *Fairness*: cuán justa es la dosis de CPU que recibe cada proceso (para alguna definición de justicia).
  - Uso de CPU: tiempo pasa la CPU siendo usada por procesos.
  - *Waiting time* (tiempo de espera): tiempo que un proceso pasa en la cola Ready (esperando CPU) a lo largo de su vida.

- *Latency* (latencia): tiempo hasta que un proceso se ejecuta por primera vez. En procesos interactivos, esto se llama *response time*.
- *Turnaround* o *completion time* (tiempo de ejecución): tiempo total que le toma a un proceso terminar.
- *Throughput* (rendimiento): número de procesos terminados por unidad de tiempo.
- Liberación de recursos: tiempo que tardan en terminar los procesos que consumen mayor cantidad de recursos.
- No se puede optimizar todos estos aspectos al mismo tiempo.
- Un scheduler puede decidir desalojar un proceso en el medio de su ejecución para darle tiempo de CPU a otro proceso.
  - Preemptive (con desalojo).
    - La CPU puede desalojar un proceso antes de su finalización, aunque él no lo haya solicitado.
    - En general, requiere un clock que interrumpa regularmente.
  - Non-preemptive (sin desalojo).
    - Los procesos sólo son desalojados cuando solicitan IO o cuando terminan.
    - Pueden ser cooperativos, en cuyo caso un proceso puede hacer llamadas explícitas para dejar que se ejecute otro proceso.
- FCFS (First-Come, First-Served):
  - Los procesos se ejecutan en el orden en que entraron a la cola Ready.
  - Cuando un proceso hace IO, sale de la CPU y vuelve al final de la cola.
  - Es non-preemptive (si admitimos desalojos compulsivos, tenemos un scheduler análogo a RR).
  - Ventajas:
    - Es fácil de implementar.
  - Desventajas:
    - Grandes waiting times. Si llega un proceso que usa mucho tiempo la CPU, los que vengan atrás van a tener que esperar ese tiempo.
    - Los procesos CPU-bounded *escoltan* a los procesos IO-bounded.
- SJF (Shortest Job First):
  - El siguiente proceso a ejecutar es aquel cuya próxima ráfaga de CPU es la más pequeña.
  - Puede ser preemptive o non-preemptive.
  - Si es preemptive, en caso de que llegue un nuevo proceso a la cola, el scheduler calcula el tiempo de su próxima ráfaga, y si es menor que lo que resta del proceso en ejecución, los intercambia.
  - Ventajas:



- Minimiza el waiting time promedio.
  - Maximiza el throughput.
- Desventajas:
  - Difícil de implementar, puesto que en general es imposible conocer el tiempo de ejecución de la próxima ráfaga de CPU de un proceso. Se puede intentar predecir, en base a la información del pasado.
  - Puede generar *starvation* (inanición). Por ejemplo, un proceso intensivo en CPU podría no ejecutarse nunca ante la llegada constante de procesos intensivos en IO.
- Priority scheduling:
  - El próximo proceso a ejecutar se elige en base a cierta prioridad.
  - Generalización de los dos anteriores.
    - FCFS toma como prioridad al tiempo de llegada.
    - SJF toma como prioridad al tiempo de la próxima ráfaga de CPU.
  - Puede ser preemptive o non-preemptive.
  - Si es preemptive, en caso de que llegue un nuevo proceso a la cola, el scheduler calcula la prioridad de este nuevo proceso, y si es mayor que la prioridad del proceso en ejecución, los intercambia.
  - Puede generar starvation.
  - Se puede mitigar la starvation introduciendo *aging*: a medida que un proceso pasa tiempo en la cola Ready, va envejeciendo y aumentando su prioridad.
- RR (Round Robin):
  - Le asigna un poco de CPU a cada proceso.
  - La duración del turno de un proceso en CPU se llama *quantum*.
    - No puede ser muy grande, porque en sistemas interactivos puede generar la sensación de falta de respuesta.
    - No puede ser muy chico, porque entonces el tiempo de scheduling + context switch (tiempo muerto en términos de procesamiento) se vuelve significativo.
    - Linux lo asigna dinámicamente, asignando pedazos de una *targeted latency*, que es el tiempo en el que espera que todas las tareas hayan sido ejecutadas al menos una vez.
  - Se suele combinar con prioridades.
    - Dadas por el tipo de usuario o indicadas por el mismo proceso (esto último no suele funcionar).
    - Que decrezca cuando un proceso recibe un quantum.

- Linux asigna un valor *nice* a cada proceso. Un proceso con *nice* bajo, es más egoísta y requiere más CPU (mayor prioridad). Cuanto más alto, más generoso y menos requerimiento de CPU (menor prioridad).
  - Ventajas:
    - Más ecuánime.
    - Más apto para sistemas que requieren tiempos de respuesta bajos.
  - Desventajas:
    - Mayor uso del tiempo en scheduling y context switching.
- Multilevel Queue Scheduling:
  - Idea: dividir a la cola Ready, en varias colas.
  - Cada cola tiene su propio algoritmo de scheduling, y además hay un algoritmo de scheduling entre las colas.
  - Un proceso siempre va a parar (y permanece) a la misma cola.
  - El scheduling entre colas suele ser de prioridades fijas y con desalojo.
- Multilevel Feedback Queue Scheduling:
  - Similar al anterior, pero ahora los procesos se pueden mover entre las colas, según su uso de CPU.
  - Si usan mucha CPU, van a parar a colas de prioridad más bajas. Ergo, los procesos interactivos van a parar a las colas de prioridad más altas.
  - Se suele combinar con aging, para que no haya procesos que permanezcan eternamente en las colas de prioridades más bajas.
- Real-Time Scheduling:
  - Para sistemas en los que los procesos tienen un tiempo de finalización (deadline) estricto.
  - En esta clase de sistemas, todas las tareas se consideran periódicas, en el sentido de que requieren usar la CPU cada  $p$  unidades de tiempo (su período).
  - Rate-Monotonic Scheduling:
    - Priority scheduler donde la prioridad es su frecuencia de ejecución  $1/p$ . A menor período, mayor prioridad.
  - Earliest-Deadline-First Scheduling:
    - Elige la tarea cuya deadline está más próxima en el tiempo.
    - Sigue la idea de que toda tarea debe terminar antes de su deadline.
- Scheduling en SMP (Symmetric Multi-Processing):
  - ¿Cómo decidimos en qué procesador ponemos a correr un proceso?
  - Preferentemente queremos mandar un proceso siempre al mismo procesador, para usar datos en cache.

- Afinidad al procesador: intento de usar siempre el mismo procesador.
  - Dura: si obligatoriamente se usa el mismo.
  - Blanda: si sólo se hace un intento de afinidad.

## Sincronización

- Hay recursos cuyo uso sólo puede ser explotado por una cierta cantidad de procesos en simultáneo. Por ejemplo, una impresora, o una instrucción.
- Por ejemplo, si dos procesos comparten una variable  $x = 0$ , y ejecutan concurrentemente la instrucción

**$x = x + 1$**

ambos podrían terminar con el valor  $x = 1$ , debido a la descomposición en instrucciones de bajo nivel

**load reg, x  
add reg, 1  
store x, reg**

- *Race condition*: el resultado de la ejecución es inválido si la ejecución se produce en cierto orden.
- Queremos impedir que dos procesos ejecuten, al mismo tiempo, el bloque de código  $x = x + 1$ . Un bloque de este tipo se llama *sección crítica*.
- Un mecanismo para resolver este problema debe satisfacer:
  - Exclusión mutua: sólo hay un proceso a la vez en la sección crítica.
  - Espera acotada: luego de que un proceso P solicita entrar en la sección crítica, hay una cantidad acotada de otros procesos que pueden entrar antes que P.
  - Progreso: la elección del próximo proceso a entrar en la sección crítica no es pospuesta indefinidamente. Además esta elección sólo depende de los procesos esperando por entrar en la sección.
- Solución 1: deshabilitar interrupciones justo antes de entrar en la sección crítica, y habilitarlas al terminar.
  - Problema: si la sección crítica es muy grande, anulamos el multitasking durante mucho tiempo. Esto es inadmisibles en sistemas interactivos.
- Solución 2: algoritmo de Peterson.
  - Para dos procesos (i es self, j es el otro).

```
while (true) {
    flag[i] = true;    // quiero entrar en la sección
    turn = j;          // pasá vos primero
    while (flag[i] == true && turn == j); // espero mientras sea
```

```

// el turno del otro
// sección crítica
flag[i] = false;    // ya no quiero entrar
// ...
}

```

- Solución 3: instrucción TestAndSet(&var) del procesador.
  - Pone 1 en var y devuelve su valor anterior.
  - Es atómica (es decir, sin permitir interrupciones durante su ejecución).
  - Idea:

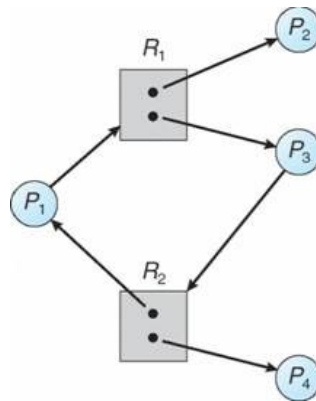
```

while (true) {
    while (TestAndSet(&lock) == 1);
    // sección crítica
}

```

- Problema de Peterson y TAS: busy waiting.
  - Busy waiting: ejecutar un while esperando a obtener el lock.
  - Este while no solo consume CPU innecesariamente, sino que probablemente lo haga por una gran cantidad de tiempo.
  - Parche: poner un sleep adentro del while. No sabemos la duración adecuada para tal sleep.
  - Mejor solución: que el SO nos despierte cuando tenemos el lock.
- Solución 4: semáforos.
  - Un semáforo es una variable entera S tal que:
    - Lo podemos inicializar en cualquier valor.
    - wait(S): decrementar S, y si  $S < 0$  esperar a ser despertado.
    - signal(S): incrementar S, y si hay procesos esperando en S, despertar alguno.
  - Un semáforo que sólo toma valores 0 y 1 se llama *mutex*.
- Todas estas soluciones al problema de la sección crítica introducen el problema de los deadlocks.
- Deadlock: un conjunto de procesos está en deadlock si cada uno de ellos está esperando un evento que sólo puede ser causado por otro proceso del conjunto.
- Condiciones de Coffman (necesarias para que haya deadlock):
  - Exclusión mutua: cada recurso está en poder, a lo sumo, de un proceso.
  - *No preemption*: no hay un mecanismo para quitarle un recurso a un proceso.

- *Hold and wait*: los procesos que ya tienen un recurso (hold) pueden solicitar y esperar por otro (wait).
  - *Circular wait*: hay una cadena de procesos en la cual un proceso está esperando algún recurso del siguiente proceso.
- Si un sistema no cumple alguna de estas condiciones, entonces está libre de deadlocks.
- Digrafo de asignación de recursos:
  - Una especie de digrafo que tiene un nodo (rectángulo) por cada proceso, y un nodo (círculo) por cada tipo de recurso.
  - Además, dentro del nodo de un tipo recurso se indica la cantidad de recursos de ese tipo que hay.
  - Hay un eje dirigido de un proceso P a un tipo de recurso R, si P está esperando por un recurso de tipo R.
  - Hay un eje dirigido de un tipo de recurso R a un proceso P, si hay un recurso de tipo R que está en manos de P.
- Una espera circular se traduce en un circuito simple en este digrafo.
  - Si hay deadlock, tiene que haber un circuito simple.
  - Sin embargo, puede haber un circuito simple en el digrafo, pero que el sistema no esté en deadlock. Por ejemplo:



- En este caso, si P4 libera su recurso, P3 lo puede tomar y se desarma el circuito.
- Problema de inversión de prioridades: si hay tres procesos con prioridades  $L < M < H$ , L está usando el recurso R, y H (que es copado) está esperando a que L termine de usarlo, podría venir M (que no es tan copado) y quitarle el recurso a L. De este modo, H tiene que esperar más tiempo para obtener R.
  - De alguna forma se “invirtió” la prioridad entre M y H.
  - Solución: protocolo de herencia de prioridades. Apenas H notifica que quiere usar R, que está en manos de L, la prioridad de L pasa a ser la misma de H. Así, M no puede venir y sacarle el recurso.

- Monitores: solución al problema de la sección crítica que no tiene deadlocks.
  - Idea: tener distintos bloques de código en los que haya exclusión mutua, cada uno con sus propias estructuras de datos.
  - Cada bloque se llama monitor.
  - Así, el único recurso que se puede pedir es el acceso a un bloque.
  - Como al pedir acceso a un bloque no podemos estar adentro de otro bloque, no se cumple hold and wait. Ergo, no hay deadlocks.
- Los monitores garantizan exclusión mutua, pero no resuelven problemas del estilo “esperar hasta que se cumpla cierta condición”.
- Variables de condición: solución al problema anterior.
  - Una variable de condición permite hacer:
    - wait(cv): esperar hasta el próximo aviso sobre cv.
    - signal(cv): despertar a un proceso esperando sobre cv.
    - broadcast(cv): avisar a todos los procesos esperando sobre cv.
  - La diferencia con los semáforos, es que las variables de condición no llevan una cuenta. Cuando se hace wait(cv), el proceso se bloquea hasta que un signal(cv) posterior lo despierte. No importa si en el pasado se hicieron otros signal(cv).
- Hay mecanismos que previenen deadlocks. Se basan en asegurar que al menos una condición de Coffman no se cumpla.
- La condición de exclusión mutua no la queremos evitar, sino todo lo contrario.
- Prevención de hold and wait:
  - Para esto, podemos hacer que un proceso obligatoriamente pida todos los recursos que va a utilizar a lo largo de su vida, al comenzar su ejecución.
- Prevención de circular wait:
  - Numeramos los recursos de la forma R1, ..., Rn.
  - Hacemos que todos los procesos pidan recursos en orden. Por ejemplo, si un proceso va a usar primero R6, luego R2 y finalmente R4, debe obligatoriamente pedir los recursos (en cualquier momento) en el orden R2, R4 y R6.
- Prevención de no preemption:
  - La idea es que si necesito recursos que están en manos de un proceso que está esperando, se los quito y los uso.
  - Es necesario que el proceso afectado pueda volver a su estado anterior fácilmente.
- Algoritmos para verificar ausencia de deadlock.
  - Otra forma de evitar deadlock es verificar, en cada solicitud de recursos, que no vaya a producir deadlock.

- Algoritmo de análisis del digrafo de asignación de recursos:
  - Cada vez que se va a asignar un recurso, se analiza dicho digrafo. Si no hay circuitos simples, entonces lo asigno, pues no hay circular wait.
- Hay otros algoritmos, que usan la noción de estado seguro
  - Estado seguro: es un estado desde el que ordenando los procesos activos de cierta forma  $P_1, \dots, P_n$ , podemos satisfacer **todas** las peticiones de recursos de  $P_1$ , liberar los recursos de  $P_1$ , satisfacer todas las peticiones de recursos de  $P_2$ , liberar  $P_2$ , y así sucesivamente hasta  $P_n$ .
  - Notar que si estamos en un estado seguro, no estamos en deadlock pues podemos asignar recursos de cierta forma de modo tal que se termine la ejecución de todos ellos.
  - El algoritmo del banquero es un ejemplo.
- Algoritmo del banquero:
  - Todos los procesos declaran, al principio de su ejecución, cuántos recursos de cada tipo van a usar a lo largo de toda su vida.
  - La idea es asegurar que, luego de cada asignación de recursos, seguimos en un estado seguro. Si esto se cumple, nunca estaremos en deadlock.
  - Cuando un proceso solicita recursos, pueden pasar dos cosas:
    - Si en este momento no hay suficientes recursos para satisfacer el pedido, denegamos la asignación.
    - En caso contrario, analizamos si luego de satisfacer el pedido, vamos a quedar en un estado seguro. En caso afirmativo, concedemos los recursos. En caso contrario, lo dejamos en espera.
    - ¿Cómo sabemos si estamos en un estado seguro?
    - Necesitamos saber si dados:
      - los recursos disponibles,
      - los recursos actualmente asignados a los procesos, y
      - los recursos que resta asignar a los procesos
 podemos ordenar a los procesos como en la definición de estado seguro.

**$P = \{P_1, \dots, P_n\}$**

**while ( $P \neq \text{vacío}$ ) {**

**Encontrar un  $P_i$  de  $P$ , tal que hay recursos disponibles como para darle todos los recursos que le faltan pedir a  $P_i$ . Si no hay un tal  $P_i$ , devolver “no es estado seguro”.**

*// podría darle todo lo que necesita a  $P_i$ , y luego que*

*// libere todo lo que tiene*

**Devolver todos los recursos que tenía asignados  $P_i$  al total de recursos disponibles.**

**$P = P - \{P_i\}$**

**}**

**devolver “es estado seguro”**

## Problemas clásicos de sincronización

- Productor-Consumidor:
  - Tenemos dos procesos, uno que produce elementos y los guarda en un buffer, mediante

**producir(buffer)**

y otro que los consume mediante

**consumir(buffer)**

- Queremos sincronizarlos, de modo tal que no se consuma lo que no se produjo, y que no se produzca al mismo tiempo que se consume (y viceversa).
- **semaphore elementos(n);**  
**mutex buffer\_mutex;**

```
productor() {  
    while (true) {  
        buffer_mutex.wait();  
        producir(buffer);  
        buffer_mutex.signal();  
        elementos.signal();  
    }  
}
```

```
consumidor() {  
    while (true) {  
        elementos.wait();  
        buffer_mutex.wait();  
        consumir(buffer);  
    }  
}
```



```

        buffer_mutex.signal();
    }
}

```

- Turnos:

- Tenemos n procesos P1, ..., Pn concurrentes. Cada proceso debe ejecutar una sentencia s<sub>i</sub>.
- Queremos que se ejecuten s1, ..., sn, en ese orden.
- **semaphore turno[n] = arreglo de n semaphore(0);**  
**turno[1].signal();**

```

P(i) {
    turno[i].wait();
    s_i;
    turno[i + 1].signal();
}

```

- *Rendezvous* (encuentro):

- Tenemos n procesos P1, ..., Pn que tienen dos partes: primero ejecutan a(i) y luego ejecutan b(i).
- Queremos que la ejecución de los b(i) comience luego de que todos hayan terminado los a(i). En otras palabras, queremos que todos los procesos se encuentren entre a(i) y b(i).
- **int cantidad = 0;**  
**mutex mutex\_cantidad;**  
**semaphore barrera(0);**

```

P(i) {
    a(i);
    mutex_cantidad.wait();
    ++cantidad;
    if (cantidad == n) {
        for (int i = 0; i < n; ++i) {
            barrera.signal();
        }
    }
    mutex_cantidad.signal();
    barrera.wait();
    b(i);
}

```

- Sección crítica de a k:
  - Queremos que a lo sumo k procesos ejecuten cierta sección crítica.
  - **semaphore sem(k);**

```

P() {
    // ...
    sem.wait();
    // sección crítica
    sem.signal();
}

```

- Readers-Writers:
  - Tenemos un recurso compartido, que puede ser leído o escrito.
  - Múltiples procesos intentan leer y escribir concurrentemente.
  - Los escritores necesitan acceso exclusivo.
  - Los lectores pueden convivir.
  - **int readers = 0;**  
**mutex mutex\_readers;**  
**mutex mutex\_recurso;**

```

writer() {
    mutex_recurso.wait();
    // escribir recurso
    mutex_recurso.signal();
}

```

```

reader() {
    mutex_readers.wait();
    if (readers == 0) {
        mutex_recurso.wait();
    }
    ++readers;
    mutex_readers.signal();

    // leer recurso

    mutex_readers.wait();
    -- readers;
    if (readers == 0) {

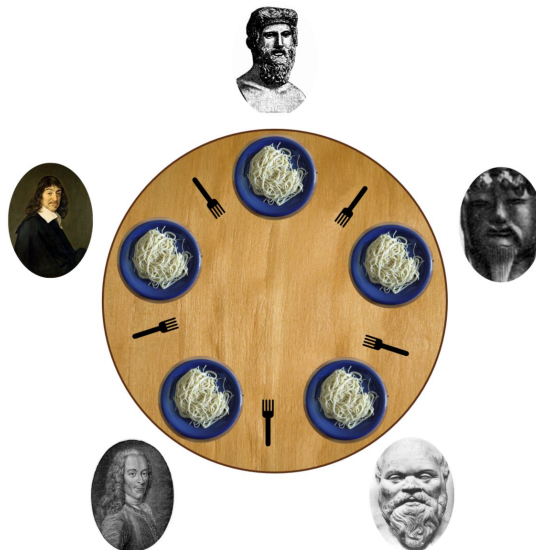
```

```

        mutex_recurso.signal();
    }
    mutex_readers.signal();
}

```

- Problema: puede haber starvation de escritores, en el caso en que no paren de aparecer lectores.
- Dining Philosophers:
  - Famoso problema, inventado por Dijkstra. No es un problema típico de sincronización, sino que es simplemente un ejemplo de deadlock.
  - Hay 5 filósofos sentados en una mesa. Entre cada filósofo hay un cubierto. Los filósofos quieren comer, y para esto deben recoger los dos cubiertos que tienen a sus costados.



- Cómo sincronizamos a los filósofos para que:
  - Todos coman (no hay deadlock ni inanición).
  - Sólo un filósofo tenga un cubierto en cada momento (exclusión mutua).
  - Más de un filósofo está comiendo a la vez (porque queremos que terminen rápido y se pongan a pensar de vuelta).
- Posible solución, numerando los filósofos del 1 al 5:
 

```

filósofo(i) {
    if (i % 2 == 0) {
        tenedores[izq(i)].wait();
        tenedores[der(i)].wait();

```

```

    } else {
        tenedores[der(i)].wait();
        tenedores[izq(i)].wait();
    }
    comer();
    tenedores[izq(i)].signal();
    tenedores[der(i)].signal();
}

```

- No hay deadlock porque no hay circular wait (probarlo por el absurdo).
- Otras soluciones:
  - Limitar los filósofos que comen de a 4. Cuando un filósofo termina de comer, dejar comer al restante.
  - Primero come el filósofo 1. Al terminar, éste le da la orden al filósofo 2. Así sucesivamente.
- Barbero:
  - Tenemos una barbería con dos salas, una de espera, con n sillas, y otra donde está la única silla en la que el barbero corta la barba.
  - Si no hay clientes, el peluquero se tira a dormir.
  - Cuando llega un cliente, si no hay lugar para esperar, se va. Si el peluquero está dormido, lo despierta.
  - **cola clientes;**  
**mutex mutexCola;**  
**semaphore sem\_clientes\_esperando(0);**

```

barbero() {
    while (true) {
        sem_clientes_esperando.wait();
        mutexCola.wait();
        sem_cliente = clientes.pop();
        mutexCola.signal();
        sem_cliente.signal();
        cortar_barba(sem_cliente);
    }
}

```

```

cliente() {
    semaphore sem_cliente;
    mutexCola.wait();
    if (clientes.size() == n) {

```

```

        mutexCola.signal();
        return;
    }
    clientes.push(sem_cliente);
    mutexCola.signal();
    sem_clientes_esperando.signal();
    sem_cliente.wait();
    // me cortan la barba
}

```

## Sistemas de archivos

- Archivo: secuencia de bytes, sin estructura.
- File system: módulo del kernel encargado de organizar la información en disco.
- Un archivo tiene muchos atributos:
  - Nombre: el nombre del archivo, posiblemente con su extensión.
  - Identificador: usado por el sistema, para identificar al archivo en el file system.
  - Ubicación: en qué volumen y qué directorio se encuentra.
  - Tamaño: en bytes o bloques.
  - Permisos: quién puede leer, escribir o ejecutar el archivo.
  - Fecha y usuario de creación y último acceso.
- Representación de archivos:
  - Un archivo se divide en bloques. Estos bloques pueden estar dispuestos de varias formas.
  - Bloques contiguos: todos los bloques son contiguos en disco.
    - Ventajas:
      - Permite lectura y escritura aleatoria rápida.
    - Desventajas:
      - Para crear el archivo necesitamos conocer cuánto espacio ocupará.
      - Fragmentación externa (más adelante se habla de esto).
      - Si el archivo crece más de lo previsto, posiblemente tenga que reacomodarlo.
  - Lista enlazada de bloques: cada bloque apunta al siguiente, y los bloques están esparcidos en el disco.
    - Ventajas:
      - Lecturas y escrituras secuenciales rápidas.
      - No hay fragmentación externa, si los bloques son de igual tamaño.

- Desventajas:
  - Lecturas y escrituras aleatorias lentas.
  - En cada bloque hace falta espacio para el puntero al siguiente bloque.
- Índice de bloques: mantener una tabla que indique la secuencia de bloques que componen un archivo. Cada bloque tiene asociada una entrada, que indica cuál es la entrada en la tabla del siguiente bloque.
  - Ventajas:
    - Las lecturas y escrituras aleatorias son rápidas si tengo la tabla cargada en memoria.
    - Los bloques no usan espacio para punteros a otros bloques.
  - Desventajas:
    - Hay que mantener toda la tabla en memoria. Si el disco es grande puede ocupar mucho espacio.
    - Hay que mantener una copia actualizada de la tabla en disco, por si el sistema se cae.
    - La tabla es única para todos los procesos, lo cual puede generar contención.
  - FAT (File Allocation Table) es un índice de bloques.
    - Los file systems de la familia FAT, dividen al disco en bloques denominados clusters.
    - La tabla FAT tiene entradas de la forma (#cluster, #del siguiente cluster en el archivo).
- La solución de Unix:
  - Cada archivo mantiene su propio índice de bloques. Cada archivo tiene un FCB (File Control Block) que guarda dicho índice. En Unix, dicho FCB se llama inodo.
  - Un inodo contiene:
    - Atributos del archivo (tamaño, permisos, timestamps, etc.).
    - Direcciones de bloques (el índice).
  - El índice se compone de cuatro tipos de entradas:
    - Direct block pointers: direcciones de bloques del archivo. En ext2 son 12.
    - Single indirect block pointers: punteros a bloques que contienen direct block pointers. En ext2 es 1.
    - Double indirect block pointers: punteros a bloques de single indirect block pointers. En ext2 es 1.
    - Triple indirect block pointers: punteros a bloques de double double indirect block pointers. En ext2 es 1.

- Ventajas:
  - Como los índices están distribuidos, puedo cargar en memoria sólo los que necesite, y además no tengo tanta contención.
  - No tengo que seguir una cadena de bloques para acceder al bloque que quiero. O sea, no es tan rápido como un acceso directo, pero tampoco es tan lento como uno completamente secuencial.
- En ext2, los bloques de datos y los bloques de inodos están separados en dos áreas distintas.

```

struct Ext2FSInode {
    unsigned short mode;
    unsigned short uid;           // owner user id
    unsigned int size;
    unsigned int atime;
    unsigned int ctime;
    unsigned int mtime;
    unsigned int dtime;
    unsigned short gid;         // owner group id
    unsigned short links_count; // cantidad de hard links
    unsigned int blocks;
    unsigned int flags;
    unsigned int os_dependant_1;
    unsigned int block[15];
    unsigned int generation;
    unsigned int file_acl;      // permisos de archivo
    unsigned int directory_acl; // permisos de directorio
    unsigned int faddr;
    unsigned int os_dependant_2[3];
}

```

- Representación de directorios:
  - Los directorios nos permiten agrupar archivos para que el usuario pueda accederlos fácilmente.
  - Un file system puede admitir distintas topologías de directorios:
    - Single level directory: único directorio (root) en el file system. Todos los archivos en dicho directorio.

- No podemos repetir nombres.
  - Todos los usuarios comparten un mismo directorio.
- Two level directory: el directorio root contiene un subdirectorio para cada usuario. Seguimos teniendo el problema de los nombres únicos.
- Tree structured directory: directorio en forma de árbol.
  - No podemos hacer que un usuario pueda acceder a archivos de otro usuario (por ejemplo, si quisieran trabajar juntos en un proyecto).
- Acyclic directory: agregamos la posibilidad de poner un link de un directorio a otro directorio o archivo.
  - Es un poquito más difícil recorrer el grafo de directorios, de modo de no repetir directorios.
  - El borrado es más complicado. Si estamos borrando un link, no debemos borrar el directorio o archivo apuntado. Si estamos borrando el directorio real, ¿qué hacemos con los links?
- General graph directory: admitimos tener ciclos entre nuestros directorios.
  - El recorrido es igual o más difícil.
- Las implementaciones tradicionales de directorios son en forma de lista enlazada y tabla de hash.
- En Unix, un directorio es una tabla de objetos de la forma (#inodo, nombre de archivo/directorio).
  - Específicamente, cada directorio tiene asociado un inodo (igual que un archivo), y las tuplas antedichas se almacenan en los bloques de datos asociados al inodo.

```

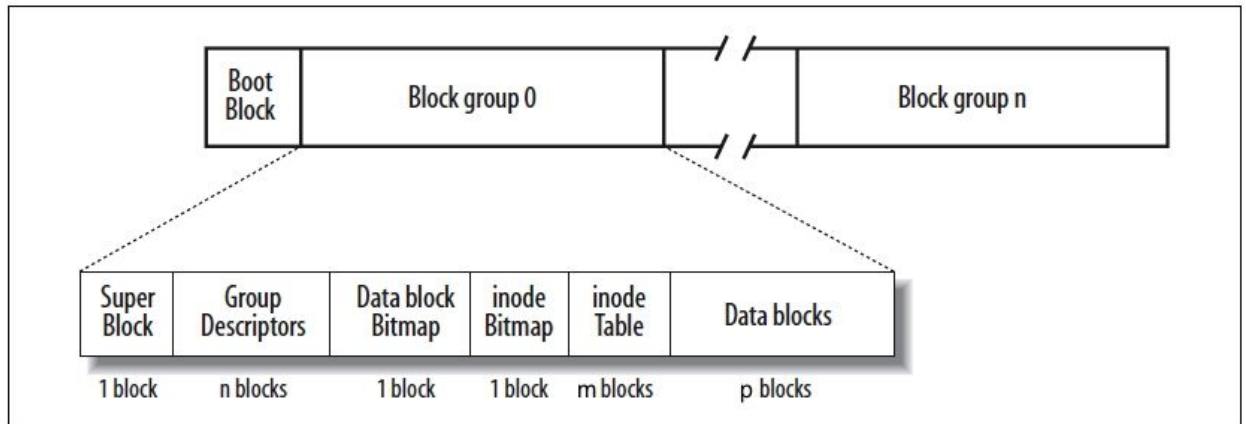
struct Ext2FSDirEntry {
    unsigned int inode;
    unsigned short record_length;
    unsigned char name_length;
    unsigned char file_type;           // con esto puedo saber si es
                                         // un archivo, directorio, char
                                         // device, block device, etc.

    char name[ ];
};

```



- Además hay un bloque llamado superblock, con un montón de información: cantidad de inodos, cantidad de bloques, tamaño de bloque, etc.



- Link: referencia a un archivo o directorio.
  - Supongamos que queremos linkear el archivo /foo/abc al archivo /bar/def (es decir, cuando acceda a /bar/def quiero acceder a /foo/abc).
  - Asumamos que el archivo abc en /foo inicialmente no existe.
  - Hard link: creamos un directory entry para abc en /foo y ponemos un puntero al inodo de /bar/def (apunto mediante el número de inodo). Incremento el contador de links del inodo de /bar/def.
  - Symbolic link: en el directory entry de abc en /foo ponemos la ruta /bar/def, indicando que debe buscarse ese archivo/directorio (apunto mediante un nombre).
  - Los inodos mantienen la cuenta de cuántos hard links hay hacia ellos.
  - Cuando se borra un symbolic link, no pasa nada.
  - Cuando se borra un hard link, el archivo se borra sólo si la cuenta llega a 0.
  - Hard links vs. symbolic links:
    - Los hard links son mucho más rápidos de resolver, ya que son un puntero a la posición física del archivo. En cambio, para resolver un symbolic link hay que resolver un nombre de archivo, para lo cual hay que recorrer el árbol de directorios.
    - Los hard links sólo pueden apuntar hacia posiciones del mismo disco. Los symbolic links pueden apuntar hacia objetos fuera del disco.
    - Los hard links permanecen válidos si el archivo apuntado cambia su ubicación. Los symbolic links quedan invalidados en este caso.

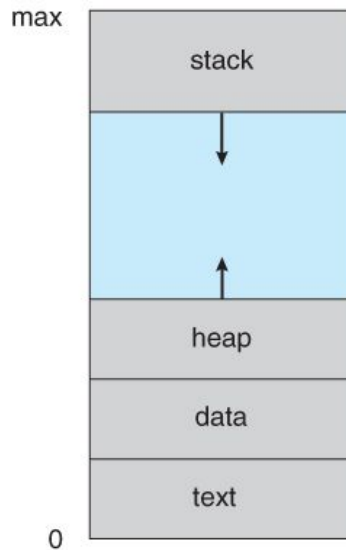
- Los hard links no pueden apuntar a directorios (para que el grafo de directorios sea un árbol). Los symbolic links sí pueden (no molestan, porque cuando recorro todo el grafo de directorios, si veo un symbolic link no lo sigo y listo).
- Manejo de espacio libre:
  - Mapa de bits: marco con 0 los bloques ocupados y con 1 los bloques libres. Para encontrar un bloque libre, busco un bit 1.
    - Su manejo es muy eficiente si la tengo almacenado en memoria. Puede ser muy grande como para tenerlo levantado en RAM.
    - Si el disco está muy vacío, el mapa de bits desperdicia mucho espacio.
  - Lista enlazada: mantengo un nodo por cada bloque libre. Para encontrar un bloque libre, tomo el primer bloque (asumiendo que todos los bloques tienen el mismo tamaño).
  - Lista enlazada clusterizada: si cada bloque puede contener N punteros a bloques, un nodo de la lista contiene N - 1 punteros a bloques libres y el N-ésimo apunta a otro nodo de la lista.
    - Un refinamiento consiste en que junto con cada puntero en un nodo se indique cuántos bloques libres consecutivos hay a partir de esa posición.
  - ext2 mantiene un mapa de bits para inodos, y otro para datos.
    - Para evitar el problema de mapas de bits extremadamente grandes, ext2 usa *group blocks*, que son un conjunto de un superblock, los mapas de bits de memoria, los bloques de inodos y los bloques de datos.
    - Un mapa de bits de un group block tiene un tamaño manejable.
- Caché:
  - Para mejorar el rendimiento se cachean bloques de disco.
  - SO modernos unifican esta caché con la de páginas, puesto que si no cada vez que traemos información de disco vamos a estar duplicando la información.
- Consistencia:
  - Ante un corte del suministro eléctrico, perdemos toda la información en caché, no bajada a disco.
  - La syscall `fsync()` graba las páginas sucias de un archivo en caché, en disco.
  - De todos modos, el sistema podría morir en cualquier momento, y con él la consistencia del file system. Para esto existe la utilidad `fsck`.

- fsck recorre todo el disco, y por cada bloque cuenta cuántos inodos le apuntan y cuántas veces aparece referenciado en la lista de bloques libres. En base a esta información toma acciones correctivas, si se puede.
- Se le agrega al FS un bit que indica apagado normal. Si al iniciarse el sistema dicho bit no está prendido, se corre fsck.
  - Correr fsck puede tomar mucho tiempo.
  - Alternativas: soft updates, journaling.
- Journaling:
  - La idea es mantener un log de las operaciones a realizar.
  - Cada conjunto de operaciones con cierta finalidad específica se denomina transacción.
  - Antes de comenzar una transacción, se escribe la lista de operaciones en disco (commit de la transacción).
  - Se mantiene un puntero sobre la transacción, que indica hasta qué punto la transacción está completada.
  - Una por una se realizan las operaciones indicadas en la misma, y a medida que se terminan las operaciones se va actualizando el puntero.
  - Si el sistema falla y una transacción no está completa, se ejecutan todas las acciones a partir de donde indica el puntero.
  - El log se almacena en disco como un buffer circular, y su escritura y lectura es eficiente, puesto que siempre es secuencial.
- Características avanzadas de un FS:
  - Cuotas de disco: cuánto espacio puede utilizar cada usuario.
  - Encriptación.
  - Snapshots: fotos del disco en determinado momento. El SO hace snapshots incrementales, duplicando sólo los archivos que se modifican. Muy bueno para hacer backups.
  - RAID por software: independencia del proveedor del hardware RAID, pero más lento.
  - Compresión.
- Performance de un FS:
  - Tecnología de disco.
  - Política de scheduling de disco.
  - Tamaños de bloque.
  - Cachés del SO.
  - Cachés de las controladoras.
  - Manejo general de locking en el kernel.
  - FS

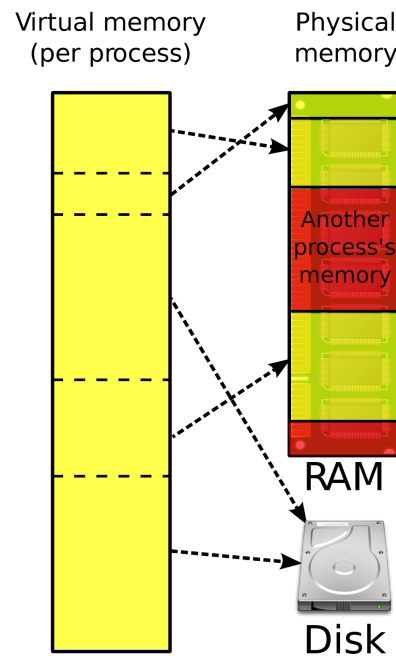
- NFS (Network File System):
  - Protocolo que permite acceder a FS remotos como si fueran locales, utilizando RPC.
  - La idea es que un FS remoto se monta en algún punto del sistema local y las aplicaciones acceden a archivos de ahí, sin saber que son remotos.
  - Para poder soportar esto, necesitamos que las operaciones sobre el FS local y remoto tengan la misma interfaz.
  - Virtual File System (VFS): interfaz entre el kernel y un FS concreto.
    - Abstrae operaciones comunes a todos los FS, de modo tal que el kernel pueda operar con él independientemente si, por ejemplo, es local o remoto.
    - Abstrae la noción de archivo local y remoto, mediante los vnodes. Si es local, un vnode se corresponde con un inodo. Si es remoto, tiene otra información.
    - En definitiva, tanto mi FS local como NFS, implementan VFS.
  - De este modo, los pedidos de IO que llegan al VFS son despachados al FS local, o al cliente de NFS, que maneja el protocolo de red necesario.
  - Observar que el cliente del NFS debe ser un módulo del kernel, porque mi FS hace la llamada al NFS, corriendo en nivel kernel. El servidor del NFS puede ser un programa en nivel usuario.
- NFS no es completamente distribuido porque los datos de un mismo directorio deben vivir físicamente en el mismo lugar.

## Administración de memoria

- La MMU (Memory Management Unit) es el módulo del sistema operativo encargado del manejo de memoria.
- Memoria física: espacio de memoria real. Cada dirección se corresponde con una ubicación física en la memoria RAM.
- Memoria virtual: espacio de memoria abstracto que utilizan los procesos.
  - Abstraerse del lugar físico permite que un programa sea independiente del lugar en memoria desde donde se ejecute, haciéndolo portable.
  - Visión de un proceso: mi memoria es un gran arreglo que empieza en la posición 0.
- Cada vez que un proceso desea acceder a una dirección de memoria (virtual), la MMU se encarga de mapearla a una dirección de memoria física.
- Espacio de memoria **virtual** de un proceso:



- Esta idea de memoria virtual vs. física se puede extender, haciendo que cada dirección de memoria virtual se pueda corresponder no sólo con direcciones en RAM sino también con disco.



- Podemos mapear varias direcciones virtuales a la misma dirección física. Esto hace que la memoria virtual puede ser mucho más grande que RAM y disco juntos.
- Podemos tener en ejecución muchos más procesos de los que entran en memoria.

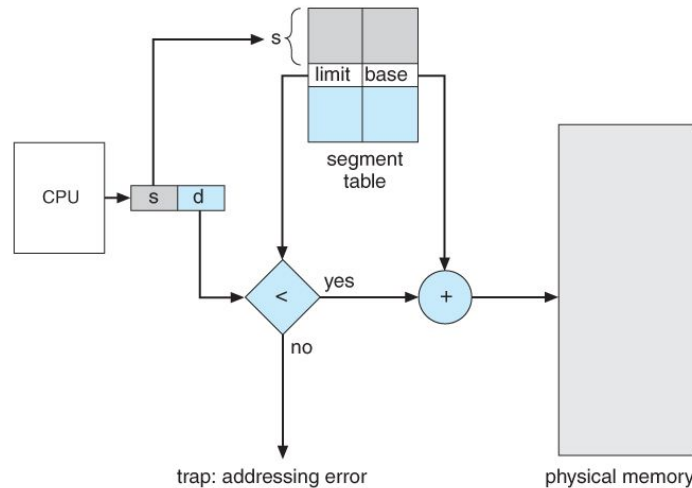
- Necesitamos poder pasar procesos de memoria a disco y viceversa.
- Swapping: pasaje de un proceso entre memoria y disco.
  - Pasos:
    - El scheduler elige un proceso de la cola ready, y le pide al dispatcher que lo ponga a ejecutar.
    - El dispatcher se da cuenta (por medio de la MMU) que el proceso está en disco.
    - Si (la MMU indica que) hay espacio suficiente en memoria, lo lee de disco y lo carga.
    - Si no, elige un proceso en memoria y lo copia a disco (y la MMU actualiza el estado de dicho proceso). Finalmente copia el nuevo proceso en ese espacio de memoria (y la MMU actualiza el estado de dicho proceso).
  - Problema: ¿qué pasa si un proceso swapeado a disco estaba esperando una respuesta de IO?
    - Solución 1: nunca swapear procesos esperando respuesta de IO.
    - Solución 2: bufferear la respuesta en espacio de memoria del SO, y luego, cuando el proceso sea swapeado a memoria, copiar el buffer al espacio del proceso.
- La forma en que se haga el mapeo depende de cómo se dispongan efectivamente las cosas (el código de los procesos y sus datos) en RAM y disco.
- La protección es otra preocupación central de todos los esquemas de asignación.
  - ¿Cómo prevenimos que un proceso acceda a memoria que no es propia?
- Asignación de memoria contigua:
  - Es la forma más básica de asignar memoria. Cuando sean necesarios N bytes, devuelvo una porción de N bytes contiguos en memoria. Este espacio reservado tiene una base (la dirección donde empieza) y un límite (por cuántos bytes se extiende).
  - Resolución de dirección virtual: hacemos base + dirección virtual.
  - Protección:
    - Si  $\text{base} + \text{virtual} \geq \text{límite}$ , el proceso está intentando acceder fuera de su espacio. Trap.
  - ¿Cómo elegimos la ubicación de la memoria a devolver?
    - Necesitamos mantener información sobre los bloques de memoria libres.
      - Bitmap del estado de cada pedacito de la memoria.
      - Lista enlazada de bloques libres.
    - Asignación de un bloque:

- Bitmap [ : ( ]: hay que buscar secuencialmente hasta encontrar una secuencia de pedacitos de memoria libre.
  - Lista [ : D ]: buscamos un bloque libre adecuado. Como sólo guarda los bloques libres, es más rápido que el bitmap.
- Liberación de un bloque:
  - Bitmap [ : D ]: simplemente marco el bitmap con las posiciones liberadas.
  - Lista [ : ( ]: la lista tiene que estar ordenada para poder fusionar bloques fácilmente. Esto nos obliga a hacer un barrido lineal.
- La memoria está formada por segmentos de memoria libre intercalados entre segmentos ocupados. Es decir, está fragmentada.
  - First fit: devolvemos memoria en el primer bloque libre en la que entre.
  - Next fit: igual que first fit, pero buscando desde la posición de la última asignación.
  - Best fit: devolvemos memoria en el bloque más chico en la que entre.
  - Worst fit: devolvemos memoria en el bloque más grande en la que entre.
  - Quick fit: se mantiene una lista de los bloques libres de los tamaños más frecuentemente solicitados.
  - Buddy system (ver el bullet “Administración de memoria del kernel”).
- Simulaciones muestran que first fit y best fit son mejores que worst fit en términos de tiempo y utilización del espacio.
- Ninguno de first fit ni best fit es mejor en términos de utilización del espacio. Sin embargo, first fit suele ser más rápido.
- Fragmentación:
  - Al aplicar cualquiera de los criterios antedichos, la memoria se fragmenta a lo largo del tiempo, a punto que comenzamos a tener pequeños bloques de memoria inservibles.
  - Estos pequeños bloques que individualmente no sirven, podrían totalizar una buena cantidad de memoria, que sería utilizable si fuera contigua.
  - **Fragmentación externa:** fragmentación de la memoria libre.
  - 50-percent rule: regla estadística que indica que first fit pierde la mitad de la memoria total a causa de fragmentación externa.
  - Solución 1: compactación.

- Consiste en compactar toda la memoria utilizada, dejando sólo un gran bloque de memoria libre.
  - Es muy costoso.
- Solución 2: asignar bloques de un tamaño fijo.
  - Esto hace que nunca se desperdicie espacio, puesto que todo bloque es asignable.
  - Si necesito N bytes, pido todos los bloques que sean necesarios para cubrir esos N bytes.
  - Problema 1: ¿Y si no hay suficientes bloques libres consecutivos?
    - Estamos al horno. Memoria contigua no puede hacer nada en este caso. Segmentación y paginación solucionan este problema.
  - Problema 2: voy a desperdiciar memoria en el último de los bloques que me asignen.
    - **Fragmentación interna:** fragmentación de la memoria asignada a un proceso.
    - Este problema lo sufren todos los esquemas que asignan bloques de tamaño fijo.
- Segmentación:
  - La memoria física del usuario se divide en segmentos.
    - Por ejemplo, un segmento para el código, otro para el heap, otro para el stack, etc.
  - Segment table: guarda la información de cada uno de los segmentos.
    - Cada entrada corresponde a un segmento, y guarda la base y límite del segmento:
  - Ahora una dirección virtual tiene dos partes:
 

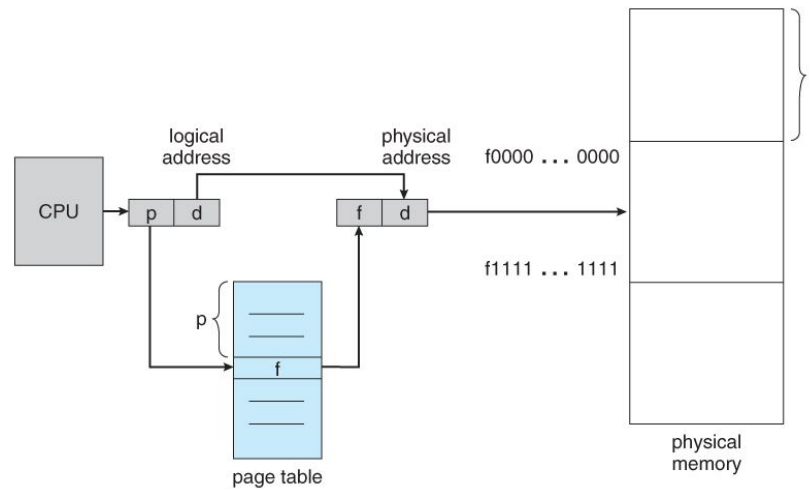
segmento : offset
  - Resolución de dirección virtual: buscamos la entrada de la tabla asociada al segmento. Luego, hacemos base + offset.





- Protección:
  - Si  $\text{base} + \text{offset} \geq \text{límite}$ , trap.
  - Además en cada entrada de la segment table guardamos bits de protección (permisos de escritura, lectura y ejecución).
- Problema de la segmentación: como los segmentos pueden tener tamaño variable, sigo teniendo fragmentación externa.
- Paginación:
  - Lo copado de la segmentación es que permite que la memoria de un proceso no necesariamente sea contigua.
  - Paginación toma esta característica, y elimina el problema de la fragmentación externa, haciendo que todos los bloques (páginas) tengan el mismo tamaño (podemos pensar a cada página como un segmento distinto).
  - La memoria virtual ahora se divide en bloques de igual tamaño, llamados páginas.
  - Una dirección virtual cae en cierto lugar de cierta página. En otras palabras, dada una dirección virtual, podemos decir su número página y offset dentro de esa página.
  - La memoria física se divide en bloques de igual tamaño, llamados frames.
  - La idea es mapear páginas de memoria virtual, a frames de memoria física.
  - Page table: indica a qué frame se mapea cada página.
  - Resolución de dirección virtual: obtenemos el número de página y offset de la dirección virtual. Vamos a la page table y buscamos el frame al que

se mapea dicha página. Finalmente, a la dirección del frame le sumamos el offset.



- Protección:
  - Un usuario sólo puede acceder a los frames mapeados desde su page table.
    - Si una página está mapeada, entonces podemos acceder al frame asociado.
    - Si no, trap.
  - Para determinar si una página está o no mapeada, se asocia a cada entrada de la tabla un bit valid-invalid. Si el bit es valid, entonces la página está mapeada. Si es invalid, no está mapeada.
  - Cada entrada de la page table también contiene bits de protección (permisos de escritura, lectura y ejecución).
- Memoria compartida con paginación:
  - Paginación resuelve fácilmente el compartimiento de memoria.
  - Para que dos procesos compartan memoria, ponemos estos datos en un frame, y hacemos que ambos procesos mapeen alguna página a ese mismo frame.
- TLB (Translation Lookaside Buffer):
  - Al introducir paginación, agregamos complejidad al proceso de resolución de direcciones virtuales.

- Necesitamos al menos dos accesos a memoria (uno a la page table y otro a la memoria física) para levantar memoria de una dirección virtual.
  - Solución: TLB
    - Es una tabla de muy rápido acceso que asocia números de página a números de marco. En algún sentido, es una caché de páginas.
- Demand paging:
  - Alternativa al swapping de procesos en sistemas con paginación.
  - En lugar de swapear procesos enteros, swapeamos páginas individualmente.
  - Más aún, sólo traemos páginas de un proceso a memoria cuando son necesarias.
  - Se hace necesario distinguir páginas en memoria de páginas en disco.
    - Se suele usar el bit valid-invalid, más campos adicionales para información sobre la dirección en disco.
    - Si el bit es valid, entonces la página está mapeada en memoria.
    - Si el bit es invalid, puede que sea porque no está mapeada, o porque está mapeada a disco.
      - Si está mapeada a disco, entonces la entrada además contiene la información sobre la dirección en disco.
- Page fault:
  - Excepción que se produce cuando un proceso intenta acceder a una página que no está mapeada en memoria (marcada como inválida).
  - Pasos:
    - El intento de acceso produce una trap.
    - El procesador busca la rutina asociada a la interrupción por page fault.
    - Se guarda el estado actual.
    - Ya sabemos que el bit valid-invalid de la entrada de la tabla de páginas era invalid. Si no hay información sobre la ubicación en disco, matar el proceso.
    - Si esa información está, encontrar un frame libre en memoria y solicitar una lectura a disco de la página buscada.
    - Mientras esperamos la finalización de la lectura, cedemos CPU a otro proceso.
    - Recibir interrupción por la finalización de la lectura.
    - Indicar a la MMU que corrija la tabla de páginas con la nueva página en memoria.

- Restablecer el estado del proceso original y continuar la ejecución.
- Copy-on-write:
  - Técnica que agiliza la creación de procesos, al evitar copiado innecesario de páginas de memoria.
  - Idea:
    - Cuando hacemos fork(), creamos un proceso copia del padre.
    - Inicialmente ambos procesos (padre e hijo) apuntan a los mismos frames.
    - Apenas uno de los dos intenta escribir alguna, se copia dicho frame a otro lugar en memoria, y ese mismo proceso cambia su mapeo. El otro sigue apuntando al viejo frame, que sigue igual que al principio.
  - Copy-on-write es útil cuando las páginas que cambia el hijo son muy pocas. Por ejemplo, cuando hacemos fork() + execv() en general no cambiamos casi nada.
    - En estos casos hay una alternativa: vfork().
    - vfork() hace que el padre quede suspendido, y que el hijo apunte a los mismos frames que el padre, pero no implementa copy-on-write. Es decir que si el hijo escribe una página, el padre podrá ver el cambio en sus páginas.
- Reemplazo de páginas:
  - Si tengo que traer una página a memoria, y la memoria está llena, tengo que elegir el frame a desalojar (victim frame).
  - Tengo que hacer 2 transferencias: una para traer la página deseada de disco, y otra para escribir la desalojada en disco.
  - Notar que si la página a desalojar nunca fue modificada desde que está en memoria, no hace falta escribirla en disco.
    - Esto se implementa con un dirty bit en cada página.
  - Si no hay frames disponibles, ¿cuál desalojo?
    - Queremos elegir un frame de modo tal de minimizar la cantidad de page faults que se producen a lo largo del tiempo.
  - Algoritmos de reemplazo:
    - FIFO:
      - Reemplazar la primer página que fue cargada en memoria.
      - Problema: no tiene en cuenta la frecuencia de uso de las páginas. No queremos desalojar páginas actualmente usadas.
    - Optimal Page Replacement:

- Reemplazar la página que no va a ser usada por el mayor lapso de tiempo.
- Garantiza mínima cantidad de page faults, dado un conjunto fijo de frames.
- Problema: en general no se cuál es esa página que dice el criterio.

■ LRU:

- Reemplazar la página que no fue usada (leída o escrita) por la mayor cantidad de tiempo.
- En general es mejor que FIFO, porque tiene en cuenta el uso de las páginas.
- Problema: es difícil de implementar.
  - Dos implementaciones clásicas son asociando un contador a cada página, o manteniendo una lista LRU. Ambas implementaciones son costosas en términos de hardware, pues tanto los contadores como la lista tienen que actualizarse en cada lectura y escritura de una página (o sea, todo el tiempo).

■ FIFO second chance:

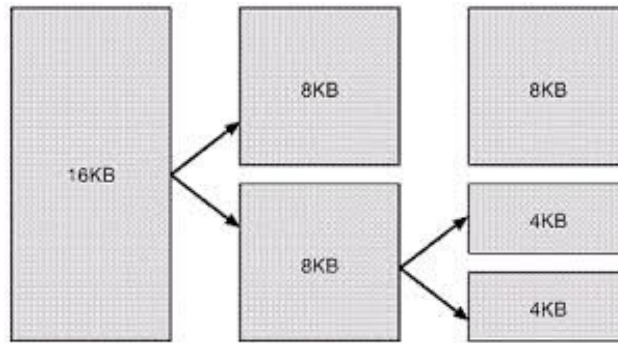
- A cada página le asociamos un reference bit. Este bit es seteado cuando una página es leída o escrita.
- Mantenemos una cola con las tareas en el orden de llegada.
  - Para desalojar una página, recorremos cada página de la cola (de la más vieja a la más nueva).
  - Si la página tiene el bit de referencia en 0, la desalojamos.
  - Si lo tiene en 1, cambiamos este bit a 0, y la reintroducimos en la cola, como si acabara de entrar.

■ Not Recently Used:

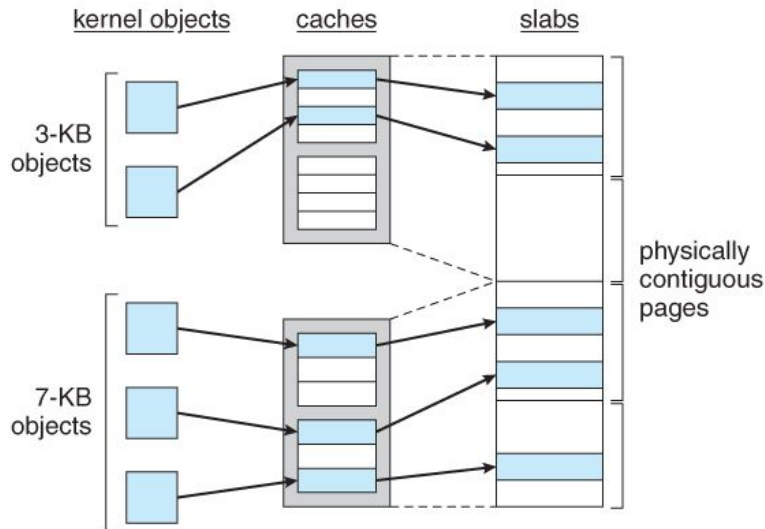
- FIFO second chance prioriza las páginas referenciadas.
- Not Recently Used prioriza en el siguiente orden:
  - Páginas modificadas y referenciadas.
  - Páginas modificadas y no referenciadas.
  - Páginas no modificadas y referenciadas.
  - Páginas no modificadas y no referenciadas.
- Desalojo una página de la categoría más baja que pueda.

- Thrashing:

- Situación en la que el SO pasa más tiempo swapeando páginas que la CPU ejecutando procesos.
- Se da cuando la memoria está llena, y hay una alta contención entre los procesos al cargar sus páginas en memoria.
  - Esto puede ser resultado, por ejemplo, de un incremento excesivo en el grado de multiprogramación (la cantidad de procesos en memoria).
- Una forma de atenuar los efectos del thrashing es mediante un algoritmo de reemplazo local.
  - Cuando un proceso necesita cargar más páginas en memoria, sólo puede desalojar sus propias páginas.
  - Esto puede ser efectivo, teniendo en cuenta que es probable que un proceso no esté usando todas sus páginas al mismo tiempo, sino sólo un subconjunto de ellas.
  - Más aún, en cada momento un proceso usa un determinado subconjunto de páginas, llamado *working set*. Utiliza intensivamente este conjunto, hasta que en algún punto lo cambia prácticamente en su totalidad. Así sucesivamente. Este comportamiento está asociado a que un programa salta de una función a otra, y durante cada una de ellas usa el mismo código y los mismos datos.
- Administración de memoria del kernel:
  - El kernel no puede usar el mismo sistema de asignación de memoria que los procesos usuario.
    - Debe usar la memoria con cuidado, puesto que su existencia debe pasar lo más desapercibida posible por los usuarios. En consecuencia, debe aprovechar al máximo el uso del espacio, y reducir al mínimo la fragmentación. Por esta razón, muchos SO no usan el mismo sistema de paginación que los usuarios.
    - Muchas veces el SO necesita que su memoria sea contigua, para poder comunicarse correctamente con dispositivos externos.
  - Buddy System:
    - Tenemos un segmento fijo de memoria contigua para asignar.
    - Al recibir el primer pedido, vamos a partir en 2 sucesivamente al segmento grande, hasta que me quede un segmentito lo más justo posible para el pedido.



- Cuando llega otro pedido, nos fijamos cuál es el segmento más chico de los disponibles. Si puedo seguir partiéndolo, y satisfacer el pedido, lo hago.
- Cada uno de los pedazos que me van quedando es un buddy.
- Cuando se libera memoria, hay que fusionar buddies mientras sea posible. Esto es fácil, porque simplemente con la posición de memoria del buddy y su tamaño, puedo saber con quién tengo que fusionarlo.
  - Coalescencia.
- Slab Allocation:
  - Slab: conjunto de frames contiguos.
  - Caché: conjunto de uno o más slabs.
  - Cada caché contendrá objetos de un mismo tipo. Por ejemplo, podría haber una caché para semáforos, otra para descriptores de procesos, etc.
  - Inicialmente, se instancian todos los objetos que puede contener cada slab de una caché.
  - Un objeto de un slab puede estar marcado como *free* (si no fue asignado) o *used* (en caso contrario).
  - Cuando una caché recibe un pedido, asignar un objeto free de una de sus slabs.
  - Cosas copadas de este sistema:
    - No hay fragmentación interna, porque cada objeto asignado tiene un tamaño prefijado, que es usado en su totalidad.
    - No hay fragmentación externa, porque puedo colocar todos los slabs pegados.
    - La asignación es muy rápida, porque ya están todos los objetos instanciados de antemano.
    - La liberación es muy rápida, porque es simplemente poner el objeto liberado en el conjunto de objetos *free* de su slab.



- API de la MMU:
  - Heap:
    - `brk()`: syscall para mover la marca del fin del heap.
      - Podemos pedir y liberar memoria así, pero no es portable.
    - `malloc()`, `free()`, `calloc`, `realloc()`: implementaciones portables para manejar asignaciones de memoria heap.
  - Stack:
    - `alloca()`
  - File mapping:
    - `mmap()`, `mmapunmap()`
    - Ideales para compartir librerías dinámicas. Varios programas haciendo llamadas al mismo código.

## Administración de E/S

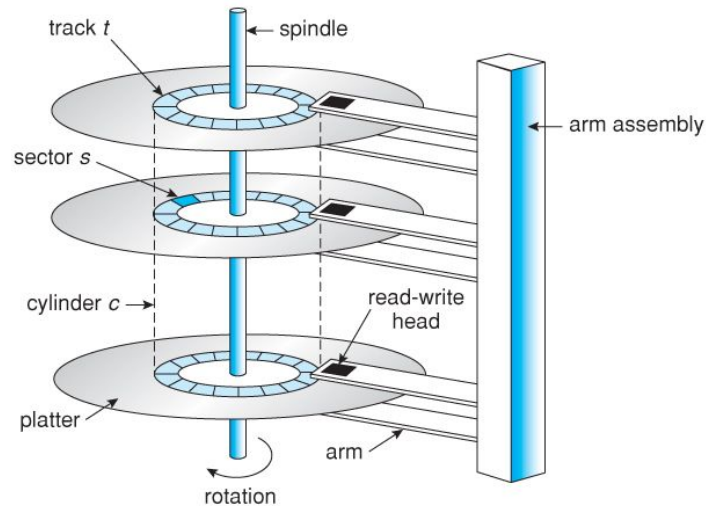
- Subsistema de E/S: módulo del SO que provee al usuario una interfaz para comunicarse con los dispositivos.
  - Provee funciones del tipo: `open()`, `close()`, `read()`, `write()`, `seek()`.
  - También provee información extra como por ejemplo si un proceso tiene acceso exclusivo a un dispositivo.
- Driver: software que provee al subsistema de E/S una interfaz para comunicarse con un dispositivo.
  - Corren con máximo privilegio.
  - Provee funciones del tipo: `driver_init()`, `driver_open()`, `driver_close()`, `driver_read()`, `driver_write()`, `driver_seek()`, `driver_remove()`
- Un dispositivo de E/S se compone de varias partes:



- Una parte electromecánica.
- Un controlador del dispositivo: una parte electrónica a la que se le puede mandar comandos mediante algún tipo de bus o registro, y nos responde de forma similar.
- En general tienen 4 registros dedicados a la comunicación con un proceso host (en la computadora a la que el dispositivo está conectado):
  - Data-in: para que el host lea input.
  - Data-out: para que el host escriba output.
  - Status: contiene bits que pueden ser leídos por el host, y que indican el estado (si cierto comando terminó de ejecutarse, si hay datos disponibles para leer en data-in, etc.)
  - Control: escrito por el host para empezar un comando o cambiar el modo de un dispositivo.
- Un puerto, que es el punto de comunicación con el host.
  - Los puertos tienen una entrada para cada registro.
  - El puerto y la CPU están conectados por un bus.
- Formas de interactuar entre un driver y un dispositivo:
  - Esta comunicación sucede a través de los registros del puerto. Hay dos formas de escribir en estos registros:
    - Ejecutando instrucciones especiales para escribir directamente en esos registros (en x86, estas instrucciones son in y out).
    - Memory mapped IO: mapeamos ciertas direcciones de RAM a ciertos registros del dispositivo. Esto nos permite escribir en dichos registros de la misma forma en que escribiríamos en memoria.
  - Polling: el driver periódicamente verifica si el dispositivo se comunicó.
    - Utilizando los registros del puerto del dispositivo, el host comunica una orden al dispositivo y lee constantemente el registro status hasta que vea que la operación se completó.
    - Ventajas:
      - Fácil de implementar.
      - No hay cambios de contexto extras.
      - No hay interrupciones asincrónicas.
    - Desventajas:
      - Consume CPU innecesariamente.
  - Interrupciones: el dispositivo avisa al driver cuando se comunicó.
    - Generalmente mediante una interrupción.
    - Esta interrupción es enviada a través de un cable que provee la CPU, llamado *interrupt-request line* (IRL), que la CPU sensa luego de ejecutar cada instrucción.

- Cuando la CPU detecta una señal en esta línea, salva el estado y salta a la rutina de atención de interrupciones indicada por la señal.
  - Ventajas:
    - Si la comunicación es poco frecuente, ahorra mucha CPU.
  - Desventajas:
    - Interrupciones asincrónicas.
- DMA (Direct Memory Access):
  - Escritura directa de un dispositivo en memoria, sin intervención de la CPU. La CPU sólo interviene al final de la escritura, cuando el dispositivo la interrumpe.
  - Se realiza a través de un controlador DMA, que forma parte del host.
    - El proceso host le indica a la CPU que quiere hacer una transferencia con DMA. Para esto debe grabar la fuente de los datos, la posición de memoria a la que desea escribirlos, y la cantidad. Todo esto se llama bloque DMA.
    - La CPU le pasa la dirección del bloque DMA al controlador DMA, y se olvida.
    - El controlador DMA hace su trabajo, ocupándose de controlar el bus de memoria para la transferencia.
  - Ventajas:
    - Permite transferir grandes volúmenes de datos sin afectar al resto del sistema.
  - Desventajas:
    - Requiere de un controlador DMA.
- Tipos de dispositivos:
  - Char device:
    - Dispositivos en los cuales se transmite la información byte a byte.
    - Ejemplos: mouse, teclado, terminales o puerto serie.
    - Soportan escritura y lectura secuencial.
    - No soportan acceso aleatorio.
    - No utilizan caché.
  - Block device:
    - Dispositivos en los cuales se transmite la información en bloque.
    - Ejemplos: disco rígido, memoria flash o CD-ROM.
    - Soporta escritura y lectura aleatorio.
    - Por lo general utilizan un buffer (caché).
- El tipo de interacción con un dispositivo depende de:

- Si el dispositivo es de lectura, escritura o lecto-escritura.
- Si permite acceso secuencial o aleatorio.
- Si es compartido o dedicado.
- Si es char device o block device.
- Si la comunicación es sincrónica (polling) o asincrónica (interrupciones).
- La velocidad de respuesta del dispositivo.
- Objetivo del SO: brindar acceso a toda la fauna de dispositivos, en forma consistente, y ocultando particularidades de cada uno, tanto como sea posible.
  - En Linux todo es un archivo.
  - El subsistema de E/S de Linux provee las siguientes funciones de alto nivel:
    - fopen, fclose: apertura y cierre de archivos.
    - fread, fwrite: leer y escribir en modo bloque.
    - fgetc, fputc: leer y escribir en modo char.
    - fgets, fputs: leer y escribir en modo char stream.
    - fscanf, fprintf: leer y escribir en modo char con formato.
    - Estas son funciones que un programa usuario puede usar para comunicarse con un dispositivo.
- Planificación E/S de disco
  - El disco es un componente presente en prácticamente toda computadora, cuya velocidad de operación es sustancialmente más lenta que el resto de los componentes.
  - Es lento porque tiene componentes mecánicos.
  - Para efectuar una lectura o escritura, hay que:
    - Mover el cabezal (para acomodarlo en el cilindro correspondiente).
    - Rotar el disco (para situar el cabezal sobre el sector correspondiente).
    - Escribir o leer.
  - La etapa más lenta es la del movimiento del cabezal.
  - Queremos minimizar esos movimientos, manejando la cola de pedidos de E/S de disco.

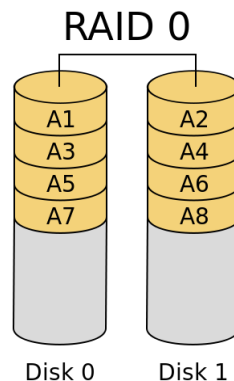


- *Seek time*: tiempo necesario para mover el cabezal.
- *Rotational latency*: tiempo necesario para rotar el disco.
- *Bandwidth*: cantidad de bytes que se pueden transferir a la vez.
- FCFS:
  - Problema: no optimizamos nada. No hace ningún esfuerzo para moverse lo menos posible.
- SSTF (Shortest Seek Time First):
  - El próximo pedido que atiende es el más cercano a donde está la cabeza en este momento.
  - Problema: puede producir inanición.
- SCAN:
  - Barrer con el cabezal de un extremo al otro, cumpliendo pedidos a medida que pasa por cada cilindro. Al llegar al otro lado, el cabezal invierte su dirección y repite el proceso.
  - Es mejor que FCFS porque optimiza un poco el movimiento del cabezal, y también es mejor que SSTF porque no genera inanición.
- C-SCAN:
  - Luego de que el cabezal barre todos los cilindros y llega al otro extremo, habrá pocos pedidos cerca del extremo donde está el cabezal y muchos pedidos en el extremo opuesto.
  - Siguiendo esta idea, C-SCAN hace que el cabezal vuelva al otro extremo, pero sin cumplir pedidos. Cuando llega al otro extremo, reanuda su operatoria normal, barriendo los cilindros y cumpliendo pedidos.
- LOOK:

- Igual a SCAN, pero en lugar de llevar el cabezal hasta el extremo, termina antes si ve que no hay más pedidos en subsiguientes cilindros.
  - C-LOOK:
    - Análogo a C-SCAN pero implementando la estrategia de LOOK.
- SSD (Solid State Drive):
  - Memoria no volátil de funcionamiento 100% electrónico.
  - Ventajas:
    - Tienen mejores tiempos de lectura y escritura, pues no tienen seek time.
    - Consumen menos energía.
  - Desventajas:
    - Son más caros que los discos magnéticos.
    - Contienen transistores, y estos pueden fallar.
    - Write amplification: la cantidad de información escrita en disco es un múltiplo del tamaño lógico de la información que se quería escribir.
- Gestión del disco:
  - Formateo físico:
    - Consiste en llenar cada sector con información (en forma de un header y un trailer) que posteriormente usará el controlador del disco. Entre esa información está el número de sector y un ECC (Error-Correcting Code).
    - Cuando el controlador escribe un sector, se actualiza el ECC calculándolo a partir del área de datos del sector.
    - Cuando el controlador lee un sector, recalcula el ECC y lo compara con el valor almacenado. Si no coinciden, el sector está corrupto. A partir del ECC se puede llegar a detectar y corregir los bits de datos que están mal.
  - Formateo lógico:
    - Primero se divide el disco en grupos de cilindros. Cada grupo se llama partición. El SO podrá tratar a cada partición como si fuera un disco distinto.
    - Luego se crea un file system para cada partición.
    - A veces se deja una partición independiente del file system, llamada *raw disk* (disco crudo). Este sector es usado cuando se quiere tener pleno control sobre la forma en que se guardan los datos. Por ejemplo, como sector de swap.
  - Booteo:

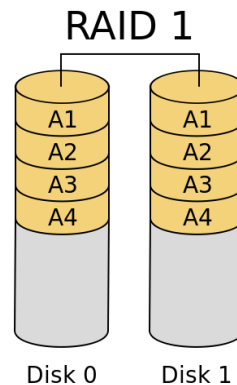
- Bootstrap: pequeño programa en ROM que permite inicializar aspectos básicos del sistema, como los registros de CPU, controladores de dispositivos y el contenido de memoria.
- En general el bootstrap se limita a inicializar lo básico, y luego carga un programa que hace la segunda parte del booteo.
- Para esto, le indica al controlador de disco que lea los bloques de booteo de disco (no hay drivers en este punto). Ya cargados en memoria, comienza la ejecución de este segundo bootstrap.
- El segundo bootstrap suele conocer la partición donde se encuentra el SO. Se encarga de cargarlo y comenzar su ejecución.
- Bloques dañados:
  - A veces, los sectores de un disco fallan y se vuelven defectuosos.
  - Distintas formas de manejarlos:
    - Escanear el disco en busca de bloques corruptos. Éstos se marcan así el FS nunca más los asigna.
    - En discos más sofisticados, el controlador mantiene una lista de los bloques corruptos. Además, mantiene información sobre bloques extra, que sólo se utilizan para reemplazar bloques dañados. Cuando un bloque falla, el controlador utiliza uno de estos bloques de sobra para hacer un reemplazo lógico del bloque roto. Esto se llama *sector sparing* o *forwarding*.
    - Notar que reemplazar un bloque por otro en otro cilindro completamente distinto impide usar las optimizaciones que haga el scheduler de disco. Por esta razón, se suelen poner bloques extra en cada cilindro.
- Spooling (Simultaneous Peripheral Operation On-Line):
  - Es una forma de manejar a los dispositivos que requieren acceso dedicado en sistemas multiprogramados.
  - Un ejemplo es la impresora.
  - La idea es designar un proceso que mantenga una cola con todos los pedidos para utilizar el dispositivo. Este proceso irá enviando las peticiones de uso a medida que el recurso se libera.
  - Los usuarios pueden ver esta cola de peticiones.
  - Es transparente para el SO, que sólo interactúa con el proceso de spooling.
- E/S para locking:
  - POSIX garantiza que `open(..., O_CREAT | O_EXCL)` es atómico. Crea el archivo si no existe, o falla si ya existe.

- Esto brinda un mecanismo sencillo, aunque no extremadamente eficiente, de exclusión mutua.
- Suele ser usado para implementar locks.
- Backups:
  - Es una copia de seguridad.
  - Tipos:
    - Total: copio absolutamente todos los datos.
    - Diferencial: copio los cambios desde el último backup total.
    - Incremental: copio los cambios desde el último backup total o incremental.
  - ¿Cómo restauro? Si la última es...
    - Total: restauro esa.
    - Diferencial: restauro la última total y luego le aplico esta diferencial.
    - Incremental: restauro la última total y luego le aplico cada una de las incrementales desde esa fecha hasta la última.
- RAID:
  - A veces no nos podemos dar el lujo de parar el sistema si un disco falla, para poder restaurar con un backup.
  - RAID (Redundant Array of Inexpensive Disks):
    - Sistemas de discos que, utilizando redundancia, pueden recuperarse de fallas.
  - RAID 0 (striping):
    - Divido todos mis datos en dos discos.
    - En general el striping es a nivel de bloque. Es decir, el bloque 0 va al disco 0, el bloque 1 va al disco 1, el bloque 2 va al disco 0, el bloque 3 va al disco 1, etc.



- Ventajas:

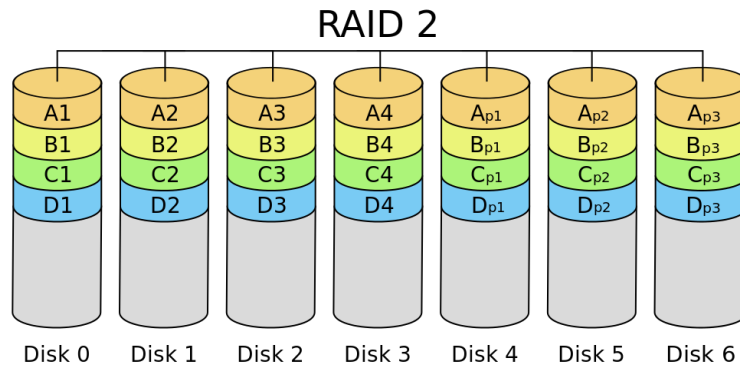
- Escrituras en paralelo, si cada disco tiene su propia controladora (es decir, si puedo darles órdenes independientemente).
- Desventajas:
  - No hay redundancia. Si falla un disco pierdo información. Ahora tengo el doble de chances de perder información.
- RAID 1 (mirroring):
  - Duplico los datos de un disco en otro disco.



- Ventajas:
  - Soporta la pérdida de uno de los discos.
  - En cada disco puedo hacer una lectura distinta, si cada disco tiene su propia controladora.
- Desventajas:
  - Tengo que escribir siempre en los dos discos.
  - Usa demasiado espacio extra para redundancia.
- Niveles siguientes:
  - Proveen redundancia mediante ECC.
  - Si falla un disco, pueden recomputar la información en él a partir del resto.
- RAID 2:
  - Usa Hamming más striping a nivel de bits.
  - Requiere 3 discos de Hamming por cada 4 de datos.
  - Soporta la pérdida de un disco.
  - Todas las lecturas y escrituras usan los 7 discos, porque todos los discos están configurados para girar con la misma orientación angular.

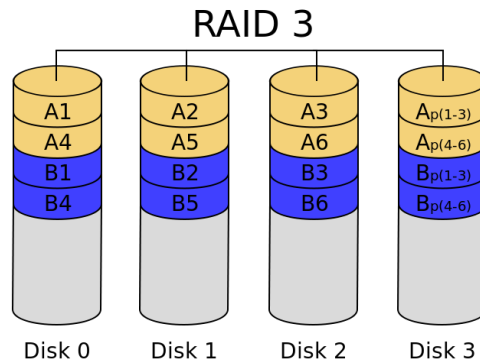


- Operan más rápido, debido a que una lectura o escritura se distribuye entre varios discos. Sin embargo, como están todos ocupados, ofrecen menos disponibilidad.



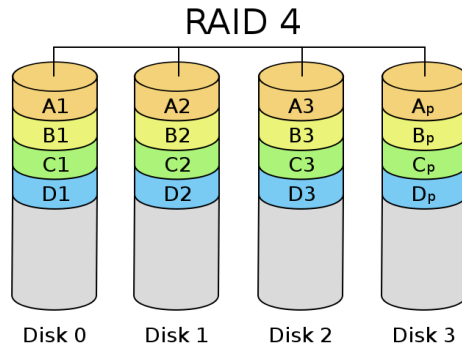
○ RAID 3:

- Usa paridad más striping a nivel de bytes.
- Requiere 1 disco de paridad por cada 4 de datos.
- Soporta la pérdida de un disco.
- Todas las lecturas y escrituras usan los 4 discos.
- Cada operación es más rápida pero hay menos disponibilidad.



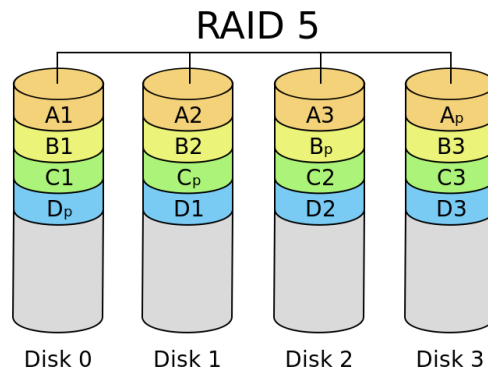
○ RAID 4:

- Similar a RAID 3 pero con striping a nivel de bloque.
- Requiere 1 disco de paridad por cada 4 de datos.
- Soporta la pérdida de un disco.
- Todas las escrituras usan el disco de paridad. Es un cuello de botella.



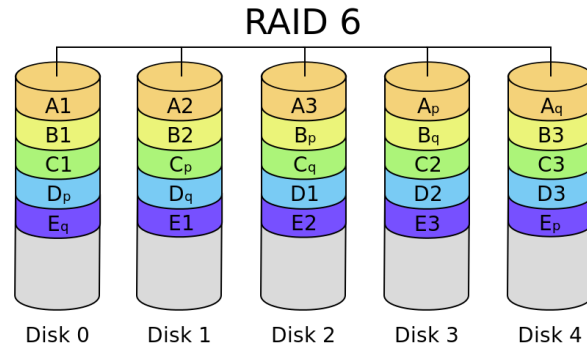
○ **RAID 5:**

- Stripping a nivel de bloque pero con paridad distribuida.
- Cada N discos en datos, requiere 1 disco más. En total quedan N + 1 discos, entre datos y redundancia mezclados.
- Soporta la pérdida de un disco.
- Todas las escrituras usan 2 discos, el de la stripe de datos y el de la stripe de paridad, pero el disco de la paridad varía. Ya no hay discos cuello de botella en las escrituras.
- La reconstrucción degrada notablemente el rendimiento.

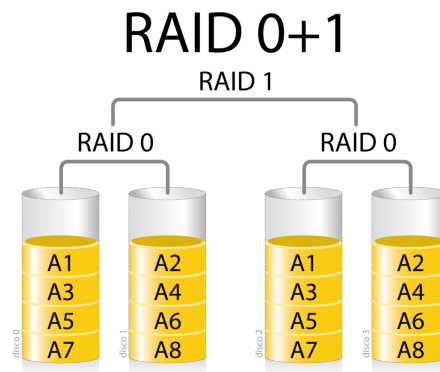


○ **RAID 6:**

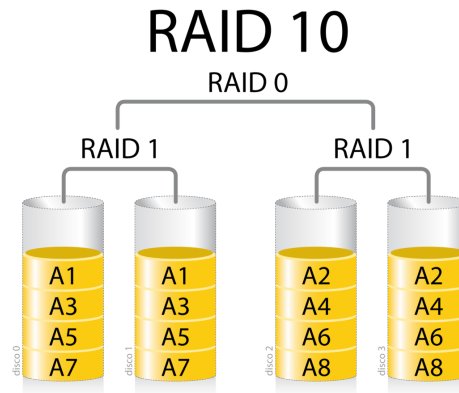
- Similar a RAID 5, pero agrega un segundo bloque de paridad, también distribuido entre todos los discos.
- Soporta la pérdida de hasta dos discos.



- RAID 0 + 1 (mirror de stripe):
  - Hago un striping de los datos. Duplico el par de discos resultantes.
  - Soporta la falla de 1 disco en un RAID 0 (un RAID 0 deja de funcionar cuando cualquiera de sus discos falla). Si hubiera una falla de otro disco en el otro RAID 0, puede que no pierda información, pero de cualquier modo el sistema deja de funcionar.



- RAID 1 0 (stripe de mirror):
  - Duplico mis datos. Hago stripes a partir del par de discos resultantes.
  - Soporta la falla de hasta 1 disco en cada RAID 1 (un RAID 1 no deja de funcionar si uno de sus discos falla).
  - Tiene un poco más de tolerancia a fallos que RAID 0 + 1, así que siempre es preferible por sobre éste.



- ¿Cómo elegir un RAID?
  - ¿Cuánto espacio en disco estoy dispuesto a perder en redundancia?
  - ¿Cuántos discos tengo disponibles?
  - ¿Puedo permitirme tiempo de downtime?
  - ¿Quiero mejorar mi velocidad de lectura/escritura?
- RAID vs. backup:
  - RAID no protege contra borrar (o modificar) un archivo accidentalmente.
  - Por eso no es sustituto de backup.
  - Backup requiere cierto downtime hasta que termine la restauración, con lo cual no es apto para sistemas real time.
  - Por eso no es sustituto de RAID.

## Protección y seguridad

- Protección:
  - Se trata de mecanismos para asegurarse de que nadie pueda acceder a los datos del otro (por ejemplo, que un proceso no pueda acceder fuera de su espacio de memoria).
  - Qué usuario puede hacer cada cosa (por ejemplo, que un proceso no pueda ejecutar cierta parte de la memoria, si no tiene permiso).
- Seguridad:
  - Se trata de asegurarse que quien dice ser cierto usuario, lo sea (por ejemplo, autenticar un usuario en un SO).
  - También se trata de impedir la destrucción o adulteración de los datos.

- Seguridad de la información:
  - preservación de:
    - Confidencialidad
    - Integridad
    - Disponibilidad
- Abstracciones útiles:
  - Un sistema de seguridad suele tener:
    - Sujetos
    - Acciones
    - Objetos
  - La idea es responder: ¿qué sujetos pueden realizar qué acciones sobre qué objetos?
  - Un sujeto puede ser un objeto. Por ejemplo: un proceso.
  - Ejemplo típico:
    - Un usuario es un sujeto del SO.
      - Es dueño de objetos: archivos, procesos, memoria, conexiones, puertos, etc.
      - Puede realizar acciones: leer, escribir, copiar, abrir, borrar, imprimir, ejecutar, matar un proceso, etc.
    - Es común que los usuarios se agrupen en grupos.
      - Los grupos son sujetos del sistema de permisos.
      - Pueden actuar de la misma manera ante ciertos objetos. Por ejemplo: archivos.
    - También se puede usar otra abstracción: los roles.
      - A un usuario se le asignan roles.
      - Los roles son los sujetos que pueden hacer cosas. Por ejemplo: operador, usuario común, administrador, etc.
- Protocolo AAA:
  - Authentication:
    - Una entidad prueba su identidad ante otra entidad.
    - Consiste en la presentación de una propuesta de identidad (por ejemplo un nombre de usuario) y la demostración de credenciales que permiten comprobarla (por ejemplo contraseñas o medios biométricos).
  - Authorization:
    - Determinar qué puede hacer una entidad, en función de su identidad (qué privilegios le doy).
  - Accounting:
    - Dejar registrado qué hizo una entidad.

- **Criptografía:**
  - Rama de la matemática y computación que se ocupa de cifrar/descifrar información utilizando métodos que permitan el intercambio de mensajes de manera que sólo puedan ser leídos por las personas a quienes van dirigidos.
- **Criptanálisis:**
  - Es el estudio de los métodos que se utilizan para quebrar textos cifrados, con el objeto de recuperar la información original, en ausencia de la clave.
- **Algoritmos de encriptación simétricos:**
  - Son aquellos que utilizan la misma clave para encriptar y desencriptar. La clave sólo la deben conocer los extremos de la comunicación.
  - Ejemplos: Caesar, DES, Blowfish, AES.
- **Algoritmos de encriptación asimétricos:**
  - Usan claves distintas. Una de ellas es pública, que puede conocer todo el mundo, y la otra es privada, que sólo conoce el receptor de un mensaje.
  - Ejemplo: RSA.
- **Funciones de hash:**
  - Ejemplos: MD5, SHA1, SHA-256.
  - Es deseable que sean one-way y resistente a colisiones.
  - One-way: que no se puede obtener la preimágen a partir del valor de hash.
    - Porque, por ejemplo, dado el hash de una contraseña, no queremos que se pueda recuperar la contraseña.
  - Resistente a colisiones: que sea difícil encontrar dos elementos distintos cuyo hash sea el mismo (colisión).
    - Porque, por ejemplo, si al autenticarme se verifica el hash de mi contraseña, no quiero que haya otra contraseña distinto cuyo hash sea el mismo.
  - Son útiles para almacenar contraseñas. Conviene guardarlas de modo tal que no se puedan leer.
    - Debe usarse Salt e iterar la función de hash varias veces.
  - Salt:
    - En general, las personas eligen contraseñas parecidas o iguales, y en general todas son cadenas que tienen algún sentido (no son caracteres random).
    - ¿Qué pasaría si directamente almacenamos el hash de cada una de estas palabras? Un atacante podría venir con su lista de hashes de password típicas, y para cada password real buscar su hash en la lista.

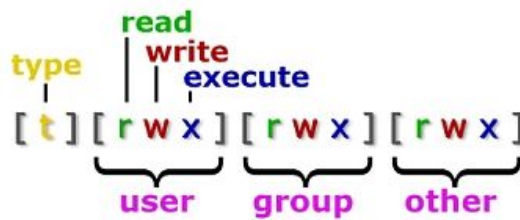
- Este ataque se basa en la existencia de contraseñas típicas. Salt intenta evitar este tipo de ataques, haciendo única a cada contraseña.
  - Para cada password  $p$  se toma una cadena al azar  $s$  (llamada salt). El hash que se almacena es  $h(s.p)$  (el de  $s$  concatenada con  $p$ ). Como  $s$  es aleatorio,  $h(s.p)$  es un hash raro que seguro no está en la lista del atacante.
  - La cadena salt no tiene que ser secreta, y se puede almacenar junto con  $h(s.p)$ . La necesitamos para verificar las contraseñas enviadas por los usuarios.
  - Notar que aún teniendo el  $s$  de cada  $p$ , un atacante no puede ejecutar su ataque tan fácilmente. Ahora, para descifrar una password  $p$ , tiene que concatenar su salt  $s$  a cada password típica  $p'$  y verificar si  $h(s.p') = h(s.p)$ .
- Iterar una función de hash:
  - Un ataque a una función de hash consiste en probar a lo bruto preimágenes.
  - Si a una password le pasamos 10000 veces una función de hash (i.e.  $h^{10000}(p)$ ), ahora el atacante para cada prueba tiene que hashear 10000 veces, lo cual hace que el ataque sea 10000 veces más lento.
- RSA (Rivest, Shamir, Adleman):
  - Es el método de encriptación asimétrica más popular.
  - Produce una clave pública, que puedo publicar, y una privada, que debo mantener secreta.
  - $D(E(m, \text{pública}), \text{privada}) = m$ ,  $D(E(m, \text{privada}), \text{pública}) = m$ .
  - Para que alguien me mande un mensaje, lo encripta usando mi clave pública y me lo manda. Solo yo lo puedo leer porque la única forma de desencriptarlo es con mi clave privada.
  - La seguridad de este método se basa en la dificultad para factorizar enteros grandes.
- Firma digital con RSA:
  - Quiero asegurar que un documento es auténtico (que nadie lo cambió).
  - Firma digital: calcula un hash del documento, y lo encripto con mi clave privada.
  - Entrego el documento más la firma digital.
  - El receptor lo desencripta con mi clave pública (está confiando que la clave pública es efectivamente la mía). Si lo puede desencriptar

exitosamente, entonces el dueño de la clave pública es efectivamente el autor del documento.

- Luego, verifica que el hash obtenido de la descripción se corresponda con el del documento.
- Autenticación remota con hash:
  - Una forma de autenticación remota de un cliente C en un servidor S podría ser la siguiente:
    - S le manda su clave pública a C que encripte su password con dicha clave.
    - C hace lo que S le pide y le manda el resultado.
    - S descripta con su clave privada y corrobora la clave de C.
  - Este procedimiento es propenso a un *Replay attack*.
    - Si un atacante M intercepta el mensaje de C a S, entonces M puede autenticarse como C.
  - Solución: *Challenge-Response*.
    - Cada vez que C se quiere autenticarse ante S, S elige un número al azar (*session token*), y se lo manda a C para que encripte su contraseña usando el número al azar como semilla.
    - C hace lo que S le pide y le manda el resultado.
    - Ahora a M no le sirve de nada interceptar ese resultado y enviárselo a S para hacerse pasar por C, porque cuando M se comunique con S, S le va a mandar otra semilla.
- Todos estos métodos criptográficos se usan para autenticar.
- Volvamos con autorización.
- Monitor de Referencias:
  - Mecanismo responsable de mediar cuando los sujetos intentan realizar operaciones sobre los objetos. Se basa en una política de acceso.
- Representación de permisos:
  - Matriz de control de accesos: matriz de Sujetos x Objetos. En las celdas figuran las acciones permitidas.
  - Detalle de implementación: se puede almacenar como una matriz centralizada, o separada por filas o por columnas. Por ejemplo, los archivos suelen guardar sólo lo que puede hacer cada usuario con ellos (es decir, la columna asociada a ese archivo).
  - Todo lo que no está dicho no se puede hacer.
  - Principio de mínimo privilegio: a cada sujeto sólo permitirle hacer lo mínimo que necesite para funcionar.
  - ¿Qué permisos le damos a un objeto creado?
    - En general estarán dados por el tipo del objeto.



- La matriz también puede ser de Sujetos x (Objetos + Sujetos), puesto que los sujetos también pueden tener permisos sobre cómo actuar sobre otros sujetos.
- DAC (Discretionary Access Control):
  - El dueño debe definir explícitamente los permisos de acceso a un objeto.
- MAC (Mandatory Access Control):
  - Cada sujeto tiene un grado.
  - El grado de un objeto es el grado del último sujeto que los modificó.
  - Un sujeto sólo puede acceder a objetos de grado menor o igual que el de él.
- DAC en UNIX:
  - Para acceder a los permisos de cada archivo:  
ls -l <archivo>
  - Devuelve la ACL (Access Control List) del archivo:



- user es el dueño del archivo.
- type:
  - - : archivo
  - d : directorio
  - l : symlink
  - c : char device
  - b : block device
  - p : pipe (un pipe registrado en la tabla de archivos).
- Podemos agregar o sacar permisos con:
 

```
chmod <u/g/o><+/-><r/w/x> <archivo>
```
- uid (user id) y gid (group id):
  - Todos los usuarios tienen un id. Lo podemos averiguar con:
 

```
id -u <nombre de usuario>
```
  - Todos los grupos tienen un id.
 

```
id -g <nombre de grupo>
```
  - Podemos ver toda la identificación de un usuario con:
 

```
id <nombre de usuario>
```
- Todo archivo tiene un owner user y un owner group.

- En general, un proceso hereda el uid del los del usuario que ejecutó el archivo.
  - A veces queremos ejecutar un proceso con mayores privilegios. Por ejemplo para cambiar la contraseña de mi usuario. Queremos ejecutar con los privilegios de root, es decir, del owner del programa. Notar que esta no es una forma de transformar al usuario en root (esto sería un obvio problema de seguridad), sino que sólo estoy ejecutando un programa de root, como root. Root escribió este programa.
  - En definitiva, un proceso tiene dos tipos de uid:
    - uid real: uid del usuario que ejecuta un proceso.
    - uid efectivo: uid del owner del archivo.
  - Análogamente, hay gid real y gid efectivo.
  - Podemos hacer que un proceso ejecute con privilegios del uid efectivo, activando el bit SETUID:
 

```
chmod u<+/->s <archivo>
```
  - Análogamente, podemos activar SETGID:
 

```
chmod g<+/->s <archivo>
```
- Sticky bit: bit de privilegio que se le puede asignar a directorios para evitar que usuarios borren archivos de otros usuarios en ese directorio.
- chattr es un comando que permite setear otros atributos de acceso (append only, immutable, etc.).
- Hay otras ACLs que son un poco más flexibles, y permiten dar permisos a usuarios específicos, a más de un grupo, etc.
- MAC en Windows:
  - Windows Integrity Control.
  - Se basa en el modelo Biba de control de integridad.
  - Define 4 niveles de integridad: System, High, Medium, Low.
  - Archivos, carpetas, usuarios, procesos, todos tienen niveles de integridad.
  - El nivel medio es el nivel por defecto para usuarios estándar y objetos sin etiquetas.
  - Un usuario no puede darle a un objeto un nivel de integridad más alto que el suyo.
- Buffer overflow:
  - Generar overflow en un buffer en stack de una función para cambiar sectores prohibidos de la memoria.
  - Típicamente se usa para cambiar el IP, que está escrito en la base del stack.
  - También se puede hacer con el heap, llenándolo hasta pisar el IP.

- Para evitarlo se pueden hacer chequeos de límites en los accesos a las estructuras de datos (esto es lo que hacen, por ejemplo, Java o C#), o chequear que no se esté accediendo a sectores prohibidos del stack.
- Control de parámetros:
  - Pasar un parámetro a un programa, que al ser utilizado genere resultados inesperados.
  - El caso típico es la SQL injection.
  - Ejemplo:
    - Programa que dada una LU marca como aprobado al alumno:

```
db.execute("UPDATE alumnos SET aprobado=true WHERE  
lu=" + input + ""
```

- Atacante ingresa:  
307/08'; DROP alumnos; SELECT '
  - El programa debería sanitizar la entrada (verificar que no sea maliciosa), y no usarla directamente.
  - Utilizar el mínimo privilegio posible.
- Condiciones de carrera:
  - Ejemplo: race condition al crear un archivo:
 

```
if (not Existe(a)) then Crear(a) fi
```

    - Lo correcto sería hacer una única operación atómica:
 

```
CrearSiNoExiste(a)
```
- Código malicioso:
  - Recibe el nombre de malware (malicious software).
  - Es software diseñado para llevar a cabo acciones no deseadas y sin el consentimiento explícito del usuario.
  - Tipos:
    - Troyano: es un software que contiene un segmento de código que se aprovecha de su entorno para realizar una acción maliciosa. En otras palabras, malware disfrazado de un programa con otras intenciones.
    - Trap door: agujero dejado en un programa a propósito por su creador. Por ejemplo, para que cierto usuario pueda loguearse en un sistema con privilegios máximos.
    - Virus: código embebido en un programa legítimo, que es capaz de replicarse a sí mismo, y que están diseñados para "infectar" otros programas (modificando o destruyendo archivos).

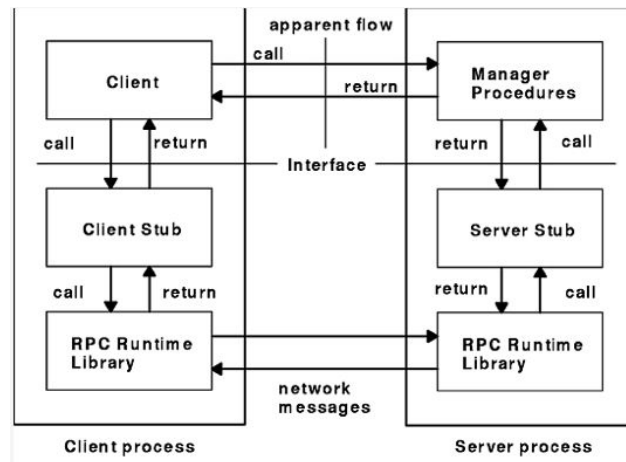
- Keylogger: guarda todo lo que se teclea.
- Principales vías de infección:
  - Descarga desde páginas web (a veces involuntariamente).
  - Adjuntos por email.
  - Vulnerabilidades en software.
  - Compartir dispositivos de almacenamiento.
  - Otros protocolos y aplicaciones de Internet: chat, P2P, redes sociales, etc.
- Cómo prevenir infecciones:
  - Usar antivirus actualizado.
  - Actualización del SO, navegador y resto de las aplicaciones.
  - Uso de usuario con privilegios restringidos.
  - Sentido común y uso responsable.
  - Mecanismos de aislamiento:
    - Sandboxes:
      - Mecanismo que provee el SO para aislar procesos.
      - Linux provee chroot.
    - Virtualización.
- Otros tipos de ataque:
  - DoS (Denial of Service):
    - Ataque que consiste en la sobrecarga del ancho de banda de la víctima o de sus recursos computacionales.
  - Privilege escalation:
    - Aprovechamiento de un bug en un sistema para ganar acceso a recursos que usualmente no son accesibles como usuario.
- Principios generales para un sistema seguro:
  - Mínimo privilegio.
  - Simplicidad.
  - Validar todos los accesos a los datos.
  - Separación de privilegios.
  - Minimizar la cantidad de mecanismos compartidos.
  - Seguridad multicapa.
  - Facilidad de uso de las medidas de seguridad.
- En general, cualquier política, mecanismo o procedimiento de seguridad está basado en asumir ciertos hechos.
  - Hay que tener claros cuáles son estos hechos.
  - Al bajarnos un parche para solucionar un problema de seguridad del SO confiamos en que:
    - El autor del parche es realmente el vendedor del SO.

- El objetivo del parche realmente es solucionar el problema, y no otra cosa.

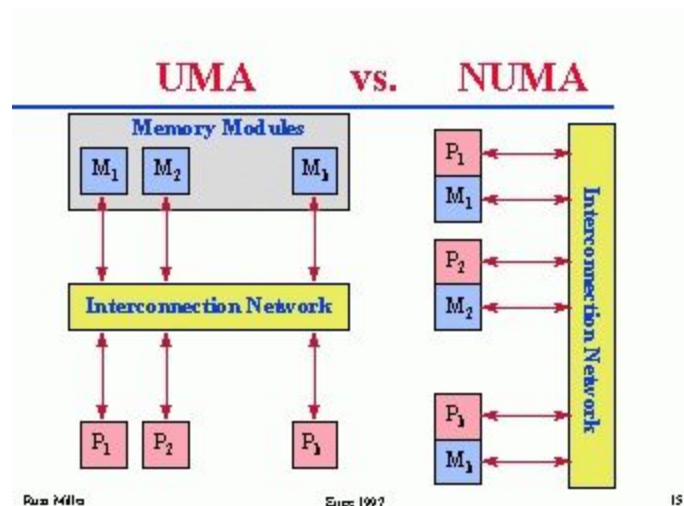
## Sistemas distribuidos

- Concurrencia: una CPU en la que se ejecutan varios procesos.
- Paralelismo: varias CPUs ejecutando procesos al mismo tiempo.
- Distribución: varias CPUs ejecutando procesos al mismo tiempo, en distintas máquinas.
- Ventajas:
  - Tiempo:
    - Reducir el tiempo de procesamiento al tener más máquinas.
  - Escalabilidad:
    - Más espacio y poder de cómputo, agregando más máquinas.
  - Tolerancia a fallas:
    - Soportar falla de una o más máquinas.
    - Replicación de datos entre máquinas.
    - Migración automática de datos entre máquinas.
- Desventajas:
  - Sincronización.
  - Consistencia.
  - Información parcial:
    - Una máquina no puede saber todo lo que sucede en el sistema.
- Paralelismo de datos vs. paralelismo de tareas:
  - De datos: aplicar el mismo procesamiento a diferentes datos.
    - Por ejemplo: paralelizar una multiplicación matricial.
  - De tareas: paralelizar distintas tareas.
    - Por ejemplo: hacerle distintos procesamientos a los frames de un video para transmitirlo eficientemente vía streaming.
- Recursos compartidos:
  - En sistemas paralelos (misma máquina)
    - Scheduler
    - Clock
    - Memoria
  - En sistemas distribuidos (distintas máquinas)
    - Canales de comunicación
    - Sólo a veces:
      - Memoria (mediante el modelo de comunicación Linda)
      - Scheduler, clock (TTA - Time Triggered Architecture)

- Comunicación:
  - Sincrónica:
    - RPC sincrónico.
  - Asincrónica:
    - RPC asincrónico.
    - Envío de mensajes (a nivel local).



- Memoria compartida:
  - Hardware:
    - Uniform Memory Access (UMA): todas las áreas de memoria son iguales de rápidas de acceder.
    - Non-Uniform Memory Access (NUMA): hay áreas más rápidas de acceder que otras para un procesador dado.



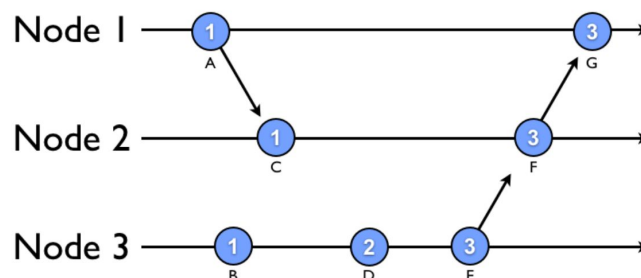
- Software:

- Estructurada:
    - Memoria asociativa (dado un valor, indican la posición).
  - No estructurada:
    - Memoria virtual global.
- Sistema de archivos:
  - Transparencia:
    - Debe poder operar de modo tal que la ubicación física de los archivos sea transparente.
  - Es el responsable de varias de las ventajas de los sistemas distribuidos:
    - Escalabilidad:
      - Permite almacenar una enorme cantidad de datos, dividiéndolos sobre los discos de muchas máquinas.
    - Tolerancia a fallas:
      - Suelen implementar replicación y migración.
- Scheduling:
  - Dos niveles:
    - Local: dar el procesador a un proceso ready. Esto pasa siempre, en cada round.
    - Global: decidir en qué procesador va a operar un proceso (mapping).
  - La asignación global puede ser de dos tipos:
    - Estática: en el momento de la creación del proceso. Define la afinidad de un proceso con un procesador. Puede ser útil en entornos NUMA.
    - Dinámica: la asignación varía durante la ejecución. Esto produce una migración.
  - La migración puede ser de dos tipos:
    - *Sender initiated*: iniciada por un procesador sobrecargado.
    - *Receiver initiated*: iniciada por un procesador libre.
  - En definitiva, la migración es una parte importante de la política de scheduling en sistemas paralelos:
    - ¿Cuándo hay que migrar un proceso?
    - ¿Qué proceso hay que migrar?
    - ¿A dónde hay que migrarlo?
    - ¿Cómo se difunde el estado?
- Conjetura de Brewer (CAP):
  - En un sistema distribuido no se puede tener a la vez:
    - Consistencia: todos los nodos ven la misma información al mismo tiempo.

- Disponibilidad: cada petición a un nodo recibe una respuesta.
- Tolerancia a partición: sigue funcionando aún si la red falla parcialmente.
- Sólo podemos obtener dos de tres al mismo tiempo.

## Comunicación por mensajes

- Problema: sincronización de procesos en un sistema distribuido, usando únicamente envío de mensajes.
  - No hay una memoria centralizada sobre la cual se puedan comunicar todos los procesos.
  - Sólo tenemos la capacidad de enviar mensajes de un proceso al resto.
- Algoritmo de Lamport para ordenar eventos:
  - Tenemos eventos que ocurren en distintos nodos de un sistema distribuido. Queremos que haya cierto orden entre ellos, para que los nodos usen este orden para sincronizar sus acciones.
  - Ordenar eventos = asignar timestamps que reflejen el orden en que ocurrieron esos eventos.
  - No queremos un reloj global, porque ello implicaría tener un nodo central dedicado a coordinarlo.
  - Es útil en algunos algoritmos que resuelven sección crítica para sistemas distribuidos.
  - La idea es que cada máquina le asocie un timestamp propio a cada evento que ocurra en el sistema.
  - Los timestamps que asignan dos procesos distintos no tienen que ser exactamente iguales, pero sí mantienen cierta relación.
  - A Lamport lo único que le importa es poder hablar del orden de los eventos en los que **seguro** podemos ponernos de acuerdo sobre su orden cronológico.





- Por ejemplo, en este caso seguro que D ocurre antes que E. También A ocurre antes que F, porque el mensaje C es la recepción de A, y C viene antes que F.
- Sin embargo, no sabemos si B ocurre antes que C o si C ocurre antes que B (aunque la imagen haga parecer que B ocurrió antes). No hay ningún evento que me permita comparar B y C. En este caso, decimos que B y C son concurrentes.
- Notación:
  - $a \rightarrow b$  si el evento a ocurre antes que el evento b.
  - $C_i(e)$  = timestamp que el proceso i le asigna al evento e.
  - $snd_i(m, t)$  = el proceso i envía el mensaje m y su timestamp actual t.
  - $rcv_i(m, t)$  = el proceso i envía el mensaje m y su timestamp actual t.
- Lo importante es que los timestamps cumplan dos cosas:
  - Si a y b ocurren en i y  $a \rightarrow b$ , entonces  $C_i(a) < C_i(b)$ .
  - Si  $e = snd_i(m, \_)$  y  $r = rcv_j(m, \_)$ , entonces  $C_i(e) < C_j(r)$ .
- Algoritmo:
  - i incrementa  $C_i$  antes de cada evento suyo.
  - i arma  $e = snd_i(m, t_i)$ , con  $t_i$  su timestamp actual. Asigna  $C_i(e) = t_i$ . Envía e.
  - j recibe  $r = rcv_j(m, t_i)$ , y actualiza  $t_j = \max\{t_i, t_j\} + 1$ . Asigna  $C_j(r) = t_j$ .
- Versión para humanos:
  - Envío:
 

```
time = time + 1;
time_stamp = time;
send(message, time_stamp);
```
  - Recepción:
 

```
(message, time_stamp) = receive();
time = max(time, time_stamp) + 1;
```
- Este algoritmo garantiza que si a ocurre (en la máquina i) antes que b (en la máquina j), entonces  $C_i(a) < C_j(b)$ .
- Si  $C_i(a) < C_j(b)$ , ¿podemos afirmar que a ocurre antes que b?
  - Seguro que b no ocurre antes que a.
  - Pero no necesariamente a ocurre antes que b.
  - Puede que sean concurrentes.
  - Pero si son concurrentes en realidad no nos importa mucho cuál ocurrió (en tiempo global) primero.

- En el algoritmo de Lamport para exclusión mutua que veremos a continuación, ésto no nos importa. Nos alcanza con saber que b no ocurre antes que a. Y **decidiremos pensar** que a ocurre (en tiempo global) antes que b b.
- Requerimientos para una solución de exclusión mutua en sistemas distribuidos:
  - [EXCL] Exclusión mutua: el recurso sólo lo puede tener un proceso a la vez.
  - Fairness: las solicitudes de uso deben ser cumplidas en el orden en que fueron hechas.
  - [LOCK-FREE] Lock-free: el recurso es concedido eventualmente (luego de tiempo finito) a **alguno** de los procesos que lo solicitó.
  - [WAIT-FREE] Wait-free: **todo** proceso que solicita el recurso lo recibe eventualmente.
  - [G-PROG] Progreso global dependiente: si todo proceso que recibe el recurso lo libera eventualmente, entonces todo proceso que solicita el recurso lo recibe eventualmente.
- WAIT-FREE => LOCK-FREE.
- Si todos los procesos hacen las cosas bien y no fallan, G-PROG = WAIT-FREE.
- Fallas en un sistema distribuido:
  - Fallas en los procesos:
    - Ningún proceso falla.
    - Los procesos caen y no se levantan.
    - Los procesos caen y se pueden levantar.
    - Los procesos caen y se pueden levantar pero sólo en determinados momentos.
  - Los procesos no son confiables:
    - Los procesos pueden comportarse de manera impredecible (fallas bizantinas).
    - Problema de los Generales Bizantinos:
      - Queremos que todos los nodos se pongan de acuerdo en algo.
      - Algunos nodos pueden fallar.
      - ¿Cómo hacemos para que los nodos sanos igual se pongan de acuerdo?
  - Fallas en la comunicación:
    - Los mensajes se pueden perder.
    - Los mensajes se pueden corromper.
  - El algoritmo de lamport para ordenar eventos sólo funciona en el caso en que ningún proceso falla y además no se pierden mensajes.

- Consenso:
  - Lograr que todos los procesos se pongan de acuerdo en algo.
- Métricas de complejidad:
  - ¿Qué cosas nos interesa medir de los algoritmos distribuidos?
    - Cantidad de mensajes que se envían.
    - Qué tipos de fallas soportan.
    - Cuánta información necesita cada nodo.
      - El tamaño de la red.
      - La cantidad de procesos.
      - Cómo ubicar a cada uno de ellos.
- Problema: exclusión mutua usando envío de mensajes.
- Solución 1: enfoque centralizado.
  - Poner el control de los recursos en un único nodo, que hace de coordinador.
  - En el coordinador hay procesos que ofician de proxies de los procesos remotos.
  - Cuando un proceso necesita un recurso, se lo pide a su proxy, quien a su vez se lo pide al coordinador.
  - Problemas:
    - El coordinador es un cuello de botella en procesamiento y capacidad de la red.
    - Si falla el coordinador, se cae el sistema.
- Solución 2: token passing.
  - Armo un anillo lógico de procesos y pongo a circular un único token.
  - Cuando quiero entrar en la sección crítica espero a que me llegue el token.
  - Desventaja: hay muchos mensajes circulando, aún cuando no son necesarios (cuando alguien no necesita entrar en la sección crítica pero igual recibe el token).
- Solución 3: algoritmo de Lamport
  - Cada proceso mantiene una cola de eventos, ordenados por el timestamp de Lamport. Cada vez que pusheo un evento en la cola, lo hago ordenadamente, según el timestamp.

### **Solicitando entrar en la sección crítica**

1. Pusheo mi solicitud REQ en mi propia cola.
2. Mando REQ a todos los procesos, con mi timestamp.
3. Espero ACK de todos los procesos (los guardo pero no en la cola).

4. Si REQ está a la cabeza de mi cola, y ya recibí todas las respuestas, entro en la sección crítica.
5. Cuando salgo, popeo REQ de la cola, y mandar un mensaje de salida REL a todos los procesos.

### **Otros procesos**

1. Cuando recibo REQ, lo pusheo. Respondo ACK.
  2. Cuando recibo REL, popeo REQ de mi cola.
  3. Si yo había mandado solicitud, vuelvo al paso 3 del algoritmo de arriba.
- Ahora sólo circulan mensajes cuando alguien quiere entrar en la sección crítica.
  - Desventaja: todos deben conocer a todos.
  - Locks distribuidos
    - Queremos obtener un lock sobre un objeto del que hay una copia en  $n$  lugares. Cada copia puede tener una versión distinta.
    - La versión naïve pide el lock a los  $n$  lugares. Se puede hacer algo mejor.
    - Para obtener un lock hay que pedirlo al menos a  $n/2 + 1$  procesos.
    - Cada sitio responde si puede o no dárnoslo.
    - Si recibimos al menos  $n/2 + 1$  objetos, leemos uno de última versión, los escribimos y actualizamos sus versiones. Para esto tomamos la versión más grande y le sumamos uno.
    - Puede haber deadlock si varios intentan pedir el lock al mismo tiempo. Podemos solucionarlo usando timestamping.
    - ¿Pueden otorgarse 2 locks a la vez? No, porque cada proceso recibe, al menos,  $n/2 + 1$  objetos.
    - ¿Puede ser que se lea y escriba una copia desactualizada (es decir, que el objeto recibido de versión más grande no sea uno de máxima versión entre todos los objetos del sistema)? No, porque siempre que alguien escribe lo hace en al menos  $n/2 + 1$  objetos.
  - Elección de líder:
    - Una serie de procesos debe elegir un líder para realizar cierta tarea.
    - Organizo los procesos en un anillo, y hago circular mi id.
    - Cuando me llega un mensaje, comparo ese id contra el mio. Hago circular el mayor.
    - Cuando un nodo detecta que el mensaje dio toda una vuelta con el mismo id, ya sabe quién es el lider.
    - Ponemos a girar un mensaje de notificación para que todos lo sepan.

- Complicaciones:
  - Nodos que fallan y se levantan.
  - Varias elecciones simultáneas.
- 2PC (Two Phase Commit):
  - Queremos realizar una transacción en todos los nodos. Todos tenemos que estar de acuerdo si se hizo o no se hizo.
  - Primera fase:
    - Un proceso coordinador envía un mensaje a cada proceso preguntando si la transacción fue satisfactoria o no. Esperar las respuestas de todos los procesos.
  - Segunda fase:
    - La transacción se considera exitosa si todos los procesos responden que fue exitosa. El resultado se informa a todos los procesos para que puedan commitear o abortar la transacción.
- Livelock:
  - Un conjunto de procesos está en livelock si estos continuamente cambian su estado en respuesta a los cambios de estado de los otros procesos del conjunto.
  - Ejemplo 1: queda poco espacio en disco. El proceso A detecta la situación y notifica al proceso B. B registra el evento en disco, disminuyendo el espacio libre, lo que hace que A detecte la situación y...
  - Ejemplo 2: tenemos poco uso de CPU por el thrashing que produce una gran cantidad de programas en memoria. El proceso A detecta el bajo uso de CPU y le informa al proceso B (scheduler) que cargue más procesos en memoria para aumentar el MPL. El proceso B carga un nuevo programa en memoria, lo cual hace que haya aún más thrashing y aún menos uso de CPU, lo cual hace que el proceso A detecta la situación y...

## Comunicación por memoria compartida

- Problema: exclusión mutua usando memoria compartida entre varias CPUs (o sea, las CPUs son locales).
- Locks
  - Lo podemos hacer teniendo operaciones atómicas para manipular mutexes:
    - `atomic Set(&mutex, valor)`
    - `atomic Get(mutex)`
    - `atomic GetAndSet(&mutex, valor)`

- atomic TestAndSet(&mutex)
- Spin-wait sobre TAS:
 

```
void lock() {
    while (TestAndSet(&mutex)) ;
}
void unlock() {
    Set(&mutex, 0);
}
```
- En lugar de prender el mutex en cada iteración, sólo hago Get:
 

```
void lock() {
    while (true) {
        while (Get(mutex)) ;
        if (!TestAndSet(&mutex)) return;
    }
}
```
- Esta última versión se llama TTAS.
- Es mejor que hacer TAS en todas las iteraciones, pero igual sigo teniendo mucho overhead por el busy-waiting.
- ¿Puedo garantizar exclusión mutua sin la operación atómica TAS?
- Modelo para razonar sobre accesos concurrentes a memoria:
  - Objeto atómico básico: *read-write register*.
  - Características:
    - Cantidad de escritores concurrentes (uno/muchos).
    - Cantidad de lectores concurrentes (uno/muchos).
    - Clasificación según la garantía que ofrece sobre el resultado de una lectura que se solapa con una escritura:
      - Safe: ninguna garantía. La lectura puede devolver fruta.
      - Regular: puede devolver el último valor o el nuevo. Más aún, si dos lecturas se solapan con una misma escritura, las dos podrían devolver distintas cosas.
      - Atomic: no hay solapamiento. En otras palabras, podemos trazar una línea de tiempo y en cada punto a lo sumo ocurre una operación.
- Los siguientes algoritmos solucionan el problema de la exclusión mutua para n procesos. Ya habíamos visto uno para 2 procesos (el algoritmo de Peterson).
- Solución 1: algoritmo de Dijkstra.
  - Usa los siguientes registros:
    - flag[i]: atomic single-writer / multi-reader
    - turn: atomic multi-writer / multi-reader

```

// flag[i] == 0 si el proceso terminó de usar la sección crítica
// flag[i] == 1 si la quiere usar
// flag[i] == 2 si cree que es su turno de usarla
P(i) {
    while (true) {
        L: flag[i] = 1;
        while (turn != i)
            if (flag[turn] == 0) turn = i;
        flag[i] = 2;
        // si hay otros procesos que también creen que es su turno
        // el turno, volvemos arriba y nos fijamos otra vez
        foreach (j != i)
            if (flag[j] == 2) goto L;

        // sección crítica

        flag[i] = 0;
    }
}

```

- Garantiza EXCL, LOCK-FREE pero no G-PROG.
- Solución 2: Panadería de Lamport.
  - Usa los siguientes registros:
    - choosing[i], number[i]: atomic single-writer / multi-reader

```

P(i) {
    while (true) {
        choosing[i] = 1;
        number[i] = 1 + max {number[j]: j != i};
        choosing[i] = 0;
        foreach (j != i) {
            waitfor (choosing[j] == 0); // que j haya sacado número
            waitfor (number[j] == 0 || // que j haya salido de la
                // panadería
                (number[i], i) < (number[j], j)); // que yo tenga
                // número más chico
        }
    }
}

```

```

    }

    // sección crítica

    number[i] = 0;
}

```

- Idea:
  - Primero todos los procesos sacan número. Cada uno trata de sacar un número distinto a todos los demás. Como están todos haciéndolo al mismo tiempo, probablemente lo hagan mal y varios terminen con el mismo número.
  - La idea es que el primero es atendido (es decir, que usa la sección crítica) es aquel con menor número. Como vimos que podía haber repetidos, van a desempatar por el número de proceso.
  - El foreach itera sobre todos los procesos, y la idea es salir de ese ciclo cuando todos los procesos con número más grande hayan sido atendidos.
  - Primero espera a que el j-ésimo proceso saque número (capaz que el j-ésimo se demoró en terminar de sacar número, pero aún así tiene el más chico).
  - Luego espera a que tenga número más chico que el proceso j, o bien a (en caso contrario) que j termine de ser atendido.
- Asegura EXCL, LOCK-FREE y G-PROG.
- Problema: el number[i] puede crecer indefinidamente.
- Hay versiones que usan contadores acotados.
- Resumen de algoritmos para el problema de exclusión mutua:
  - Registros atomic multi-writer / multi-reader:
    - EXCL, LOCK-FREE pero no G-PROG:
      - Dijkstra
    - EXCL, LOCK-FREE y G-PROG:
      - Peterson
      - Tournament
  - Registros atomic single-writer / multi-reader:
    - EXCL y LOCK-FREE pero no G-PROG:
      - Burns
    - EXCL, LOCK-FREE y G-PROG:
      - Lamport (Panadería)
- Todos los algoritmos que vimos requieren  $O(n)$  registros read-write.



- Teorema (Burns & Lynch). No se puede garantizar EXCL y LOCK-FREE con menos de  $n$  registros read-write.
- Se puede hacer algo mejor pero asumiendo restricciones de tiempo en el modelo (es decir, que ciertas operaciones no toman más de cierto tiempo). Algoritmo de Fischer.
- Algoritmo de Fischer:
  - Usa los siguientes registros:
    - `turn`: multi-writer / multi-reader

```

P(i) {
    while (true) {
        L: waitfor (turn = 0);
        turn = i;      // asume que tarda a lo sumo d
        pause D;      // asume D > d
        if (turn != i) goto L;
        // sección crítica
        turn = 0;
    }
}

```

- Garantiza EXCL y LOCK-FREE si  $D > d$ .
- La asunción  $D > d$  es una forma de esperar suficiente tiempo para que todos hagan `turn = i`. Entonces, el último que pide el turno se lo queda.
- Teorema (Herlihy & Lynch). No se puede garantizar consenso con registros read-write atómicos.
- Esto implica que no se pueden implementar locks con registros read-write atómicos en un sistema distribuido.
- Jerarquía de objetos atómicos (Herlihy).
  - *Consensus number*: cantidad de procesos para los que resuelve consenso un objeto.
  - Se sabe el consensus number de varios objetos:
    - Registros read-write atómicos: 1
    - Colas, pilas: 2
    - `TestAndSet()`: 2
    - `CompareAndSwap(expected, new)`: infinito
  - `CompareAndSwap` compara el valor actual con `expected`. Si es igual, lo reemplaza por `new`.